

DDDDDDDDDDDD	EEEEEEEEEEEEEE	BBBBBBBBBBBBBB	UUU	UUU	GGGGGGGGGGGG
DDDDDDDDDDDD	EEEEEEEEEEEEEE	BBBBBBBBBBBBBB	UUU	UUU	GGGGGGGGGGGG
DDDDDDDDDDDD	EEEEEEEEEEEEEE	BBBBBBBBBBBBBB	UUU	UUU	GGGGGGGGGGGG
DDD	DDD	BBB	UUU	UUU	GGG
DDD	DDD	BBB	UUU	UUU	GGG
DDD	DDD	BBB	UUU	UUU	GGG
DDD	DDD	BBB	UUU	UUU	GGG
DDD	DDD	BBB	UUU	UUU	GGG
DDD	DDD	BBB	UUU	UUU	GGG
DDD	DDD	BBB	UUU	UUU	GGG
DDD	DDD	BBB	UUU	UUU	GGG
DDD	DDD	BBB	UUU	UUU	GGG
DDD	DDD	BBB	UUU	UUU	GGG
DDD	DDD	BBB	UUU	UUU	GGG
DDD	DDD	BBB	UUU	UUU	GGG
DDD	DDD	BBB	UUU	UUU	GGG
DDD	DDD	BBB	UUU	UUU	GGG
DDD	DDD	BBB	UUU	UUU	GGG
DDD	DDD	BBB	UUU	UUU	GGG
DDD	DDD	BBB	UUU	UUU	GGG
DDD	DDD	BBB	UUU	UUU	GGG
DDDDDDDDDDDD	EEEEEEEEEEEEEE	BBBBBBBBBBBBBB	UUUUUUUUUUUUUU	UUUUUUUUUUUUUU	GGGGGGGGGG
DDDDDDDDDDDD	EEEEEEEEEEEEEE	BBBBBBBBBBBBBB	UUUUUUUUUUUUUU	UUUUUUUUUUUUUU	GGGGGGGGGG
DDDDDDDDDDDD	EEEEEEEEEEEEEE	BBBBBBBBBBBBBB	UUUUUUUUUUUUUU	UUUUUUUUUUUUUU	GGGGGGGGGG

```
DDDDDDDD  BBBB BBBB  GGGGGGGG  PPPPPPPP  AAAAAA  RRRRRRRR  SSSSSSSS  EEEEEEEEE  RRRRRRRR
DDDDDDDD  BBBB BBBB  GGGGGGGG  PPPPPPPP  AAAAAA  RRRRRRRR  SSSSSSSS  EEEEEEEEE  RRRRRRRR
DD      DD  BB      BB  GG      GG  PP      PP  AA      AA  RR      RR  SS      SS  EE      EE  RR      RR
DD      DD  BB      BB  GG      GG  PP      PP  AA      AA  RR      RR  SS      SS  EE      EE  RR      RR
DD      DD  BB      BB  GG      GG  PP      PP  AA      AA  RR      RR  SS      SS  EE      EE  RR      RR
DD      DD  BBBB BBBB  GG      GG  PPPPPPPP  AA      AA  RRRRRRRR  SSSSSS  EE      EE  RRRRRRRR
DD      DD  BBBB BBBB  GG      GG  PPPPPPPP  AA      AA  RRRRRRRR  SSSSSS  EE      EE  RRRRRRRR
DD      DD  BB      BB  GG      GG  PP      PP  AA      AA  RR      RR  SS      SS  EE      EE  RR      RR
DD      DD  BB      BB  GG      GG  PP      PP  AA      AA  RR      RR  SS      SS  EE      EE  RR      RR
DD      DD  BB      BB  GG      GG  PP      PP  AA      AA  RR      RR  SS      SS  EE      EE  RR      RR
DD      DD  BB      BB  GG      GG  PP      PP  AA      AA  RR      RR  SS      SS  EE      EE  RR      RR
DDDDDDDD  BBBB BBBB  GGGGGG  PP      PP  AA      AA  RR      RR  SSSSSSSS  EEEEEEEEE  RR      RR
DDDDDDDD  BBBB BBBB  GGGGGG  PP      PP  AA      AA  RR      RR  SSSSSSSS  EEEEEEEEE  RR      RR
.....
```

```
LL      LL      SSSSSSSS
LL      LL      SSSSSSSS
LL      LL      SS
LL      LL      SS
LL      LL      SS
LL      LL      SS
LL      LL      SSSSSS
LL      LL      SSSSSS
LL      LL      SS
LL      LL      SS
LL      LL      SS
LLLLLLLLLL  IIIIII  SSSSSSSS
LLLLLLLLLL  IIIIII  SSSSSSSS
```

```
1 0001 0 MODULE DBGPARSER (IDENT = 'V04-000') =
2 0002 0
3 0003 1 BEGIN
4 0004 1
5 0005 1 *****
6 0006 1 *
7 0007 1 *   COPYRIGHT (c) 1978, 1980, 1982, 1984 BY   *
8 0008 1 *   DIGITAL EQUIPMENT CORPORATION, MAYNARD, MASSACHUSETTS. *
9 0009 1 *   ALL RIGHTS RESERVED.                  *
10 0010 1 *
11 0011 1 *   THIS SOFTWARE IS FURNISHED UNDER A LICENSE AND MAY BE USED AND COPIED *
12 0012 1 *   ONLY IN ACCORDANCE WITH THE TERMS OF SUCH LICENSE AND WITH THE *
13 0013 1 *   INCLUSION OF THE ABOVE COPYRIGHT NOTICE. THIS SOFTWARE OR ANY OTHER *
14 0014 1 *   COPIES THEREOF MAY NOT BE PROVIDED OR OTHERWISE MADE AVAILABLE TO ANY *
15 0015 1 *   OTHER PERSON. NO TITLE TO AND OWNERSHIP OF THE SOFTWARE IS HEREBY *
16 0016 1 *   TRANSFERRED.                          *
17 0017 1 *
18 0018 1 *   THE INFORMATION IN THIS SOFTWARE IS SUBJECT TO CHANGE WITHOUT NOTICE *
19 0019 1 *   AND SHOULD NOT BE CONSTRUED AS A COMMITMENT BY DIGITAL EQUIPMENT *
20 0020 1 *   CORPORATION.                          *
21 0021 1 *
22 0022 1 *   DIGITAL ASSUMES NO RESPONSIBILITY FOR THE USE OR RELIABILITY OF ITS *
23 0023 1 *   SOFTWARE ON EQUIPMENT WHICH IS NOT SUPPLIED BY DIGITAL. *
24 0024 1 *
25 0025 1 *
26 0026 1 *****
27 0027 1
28 0028 1   WRITTEN BY
29 0029 1       Bert Beander      February, 1982
30 0030 1
31 0031 1   MODIFIED BY
32 0032 1       Rich Title           Added code and tables needed to
33 0033 1                               support BLISS, C, MACRO.
34 0034 1       Ping Sager          Added code and tables needed to
35 0035 1                               support PASCAL.
36 0036 1       Walter Carrell III  Update DBG$GL_CURRENT_PRIMARY for
37 0037 1                               self referential records
38 0038 1       Brad Becker         Added tables and support for
39 0039 1                               Built-in functions.
40 0040 1
41 0041 1
42 0042 1   MODULE FUNCTION
43 0043 1       This module contains the language-independent lexical scanner and
44 0044 1       parser used to scan and parse both address expressions and language
45 0045 1       expressions for all languages. It also contains all the parse tables
46 0046 1       needed to scan and parse each of the languages supported by DEBUG.
47 0047 1
48 0048 1
49 0049 1   REQUIRE 'SRC$:DBGPROLOG.REQ';
50 0183 1
51 0184 1   LIBRARY 'LIB$:DBGGEN.L32';
52 0185 1
53 0186 1   FORWARD ROUTINE
54 0187 1       AAA DUMMY: NOVALUE,           ! Dummy routine--does nothing, not used
55 0188 1       DBG$ADDR_EXP_INT,           ! Address Expression Interpreter driver
56 0189 1       DBG$BUILD_PRIMARY_SUBNODE: NOVALUE, ! Build a Primary Descriptor Sub-Node
57 0190 1       DBG$EXP_INT,               ! Expression Interpreter driver routine
```



```

: 58      0191 1   DBG$EXPRESSION_PARSER,
: 59      0192 1   DBG$GET_BIF_ARGUMENTS,
: 60      0193 1   DBG$LEXICAL_SCANNER,
: 61      0194 1   DBG$PARSER_SET_LANGUAGE: NOVALUE,
: 62      0195 1   DBG$PRIMARY_PARSER: NOVALUE,
: 63      0196 1   APPEND_TO_PATHNAME: NOVALUE,
: 64      0197 1   CHECK_OPSCOPE,
: 65      0198 1   CONSTANT_TO_VALDESCR,
: 66      0199 1   CREATE_OPERAND_TOKEN,
: 67      0200 1   CREATE_OPERATOR_TOKEN,
: 68      0201 1   CREATE_PRID_CONSTANT,
: 69      0202 1
: 70      0203 1   DUMP_OPERATOR: NOVALUE,
: 71      0204 1   DUMP_TOKEN: NOVALUE,
: 72      0205 1   DUMP_PRIMARY: NOVALUE,
: 73      0206 1   FIX_OP_PRIMARY: NOVALUE,
: 74      0207 1   GET_BLISS_SUBSCRIPTS: NOVALUE,
: 75      0208 1   GET_DEREFERENCE: NOVALUE,
: 76      0209 1   GET_FIELDREF: NOVALUE,
: 77      0210 1   GET_RECORD_COMPONENT: NOVALUE,
: 78      0211 1   GET_RECORD_VARIANT,
: 79      0212 1
: 80      0213 1   GET_SET_CONSTANT,
: 81      0214 1   GET_SUBSCRIPTS: NOVALUE,
: 82      0215 1   GET_SUBSTRING: NOVALUE,
: 83      0216 1   OPERATOR_TO_RESTORE_RADIX,
: 84      0217 1   PATHNAME_TO_PRIMARY,
: 85      0218 1   RESOLVE_COMPONENT,
: 86      0219 1
: 87      0220 1   SAVE_SUBSCRIPTS: NOVALUE,
: 88      0221 1   SCAN_QUOTED_STRING: NOVALUE;

```

```

! Parser to parse language expressions
! Parse and pick up built-in function argument list
! Lexical scanner for all languages
! Set language tables for parser
! Parser to parse a primary symbol
! Append a name to Pathname Descriptor
! Go upscope from record component
! Build Value Descriptor for a constant
! Create a Token Entry for an operand
! Create a Token Entry for an operator
! Create a Predefined Identifier Constatn
!   value descriptor
! Dump an operator being evaluated
! Dump a Token Entry symbolically
! Dump a Primary or Value Descriptor
! Fix up a Primary with subscripts
! Pick up BLISS subscripts
! Do a derefence (PASCAL ^ operator)
! Pick up field reference X<p,s,e>
! Do record component selection
! Search a set of record variants for
!   a specified record component
! Parse and pick of set constants
! Parse and pick of array subscripts
! Parse and pick up substring reference
! Returns operator to restore radix
! Construct Primary Descr from pathname
! Attempt to resolve possibly
!   ambiguous record references.
! Save away subscripts
! Scan a quoted character string

```


90	0222	1	EXTERNAL ROUTINE	
91	0223	1	DBG\$DATA_LENGTH,	Length in bits of data in VMS descriptor
92	0224	1	DBG\$DEF_SYM_FIND,	Look up DEFINEd symbol
93	0225	1	DBG\$DUMP_HEX: NOVALUE,	Dump a memory block in hexadecimal
94	0226	1	DBG\$ENUM_POS,	Convert enum val-> enum position
95	0227	1	DBG\$ENUM_VAL,	Convert enum pos-> enum val
96	0228	1	DBG\$EVAL_OP SET LANGUAGE: NOVALUE,	Sets up Operator Information Tables
97	0229	1	DBG\$EVAL_ADA_TICK,	Evaluate an Ada Tick operator
98	0230	1	DBG\$EVAL_ADDR_OPERATOR,	Evaluate an Address Expr. operator
99	0231	1	DBG\$EVAL_LANG_OPERATOR,	Evaluate a language expr. operator
100	0232	1	DBG\$GET_TEMP_MEM,	Get a temporary memory block
101	0233	1	DBG\$HASH_FIND,	Look up symbol in RST Hash Table
102	0234	1	DBG\$HASH_FIND_SETUP: NOVALUE,	Set up call on DBG\$HASH_FIND
103	0235	1	DBG\$PERFORM_TYPEID_CHECK,	Check type consistency in set constant
104	0236	1	DBG\$MAKE_SKELETON_DESC,	Make skeleton Value or Primary Descr.
105	0237	1	DBG\$MAP_DTYPE_CLASS,	Given DTYPE, do a best guess at the CLASS
106	0238	1	DBG\$NCOB_PATHDESC_TO_CS: NOVALUE,	Pathname to COBOL string
107	0239	1	DBG\$NCOPY_DESC,	Copy a descriptor
108	0240	1	DBG\$NEWLINE: NOVALUE,	Close print line and start new line
109	0241	1	DBG\$NPATHDESC_TO_CS: NOVALUE,	Convert Pathname Descr to ASCII
110	0242	1	DBG\$NUM_BYTES,	Return the length of a given dtype
111	0243	1	DBG\$PRIM_TO_ADDR,	Convert Primary to address
112	0244	1	DBG\$PRIM_TO_VAL,	Convert Primary to Value Descriptor
113	0245	1	DBG\$PRINT: NOVALUE,	Print FAO-formatted text
114	0246	1	DBG\$PRINT SET LANGUAGE: NOVALUE,	Sets up Print Information Tables
115	0247	1	DBG\$STA_GETSYMBOL: NOVALUE,	Look up a symbol in the RST
116	0248	1	DBG\$STA_SETCONTEXT: NOVALUE,	Set up current context
117	0249	1	DBG\$STA_SYMSIZE: NOVALUE,	Look up a symbol's size
118	0250	1	DBG\$STA_SYMTYPE: NOVALUE,	Look up a symbol's data type in RST
119	0251	1	DBG\$STA_SYMVALUE: NOVALUE,	Look up a symbol's value
120	0252	1	DBG\$STA_TYPEFCODE,	Get FCODE of a given Type ID
121	0253	1	DBG\$STA_TYP_ARRAY: NOVALUE,	Look up array data type information
122	0254	1	DBG\$STA_TYP_ATOMIC: NOVALUE,	Look up atomic data type information
123	0255	1	DBG\$STA_TYP_DESCR: NOVALUE,	Look up descriptor type information
124	0256	1	DBG\$STA_TYP_FILE: NOVALUE,	Get type info for file variable
125	0257	1	DBG\$STA_TYP_RECORD: NOVALUE,	Look up record data type information
126	0258	1	DBG\$STA_TYP_SUBRNG: NOVALUE,	Look up subrange data type information
127	0259	1	DBG\$STA_TYP_TYPEDPTR: NOVALUE,	Look up typed pointer information
128	0260	1	DBG\$STA_TYP_VARIANT: NOVALUE,	Look up record variant set information
129	0261	1	DBG\$STA_TYP_VARIANT_COMP: NOVALUE,	Look up components of a record variant
130	0262	1	DBG\$TYPEID_FOR_ARRAY,	Construct a typeid for an array
131	0263	1	DBG\$TYPEID_FOR_ATOMIC,	Construct a typeid for an atomic object
132	0264	1	DBG\$TYPEID_FOR_SET,	Construct a Set Constant typeid
133	0265	1	OTSS\$CVT_TI_L;	Convert ASCII string to integer
134	0266	1		
135	0267	1	EXTERNAL	
136	0268	1	DBG\$GL_ARRSUB_FLAG,	
137	0269	1	DBG\$GL_DEVELOPER: BITVECTOR[],	Developer mode control bits
138	0270	1	DBG\$GL_CURRENT_PRIMARY,	Pointer to the primary being processed
139	0271	1	DBG\$GB_LANGUAGE: BYTE,	Currently set language
140	0272	1	DBG\$GB_MOD_PTR: REF VECTOR [,BYTE],	Pointer to modes
141	0273	1	DBG\$GL_ORIG_COMMAND_PTR,	Pointer to original command string
142	0274	1	DBG\$GL_RECCMP_FLAG,	
143	0275	1	DBG\$GB_SET_BREAK_FLAG: BYTE,	Set to TRUE during parsing of
144	0276	1		SET BREAK command.
145	0277	1	DBG\$GL_UPCASE_COMMAND_PTR: VECTOR[2];	
146	0278	1		Pointers to start and end

147	0279	1		! of current command string
148	0280	1		
149	0281	1	GLOBAL	
150	0282	1	DBG\$GL_CHARTBL: REF CHRTBL\$TABLE;	! Pointer to character table for the
151	0283	1		! currently set language
152	0284	1		
153	0285	1	LITERAL	
154	0286	1	CAR_RET = %CHAR(13),	! Carriage-return character
155	0287	1	VARSTK_SIZE = 20;	! Maximum nesting depth of record
156	0288	1		! variants we allow
157	0289	1		
158	0290	1	! These are passed into APPEND_TO_PATHNAME as the third parameter. This	
159	0291	1	! parameter tells the routine how to insert the name into the pathname	
160	0292	1	! being constructed.	
161	0293	1		
162	0294	1	LITERAL	
163	0295	1	NOT_REC_COMP = 0,	
164	0296	1	REC_COMP = 1,	
165	0297	1	COB_REC_COMP = 2;	
166	0298	1		
167	0299	1	OWN	
168	0300	1	ADDRESS_LENGTH,	! Length of instruction pointed to by
169	0301	1		! address expression (or zero)
170	0302	1	ADDRESS_TYPE,	! Address expression "type"--either
171	0303	1		! "instruction" or "no type"
172	0304	1	BIF_TABLE: REF VECTOR[.LONG],	! Pointer to a Built-in Function table
173	0305	1		! for the current language where
174	0306	1		! the functions are identifiers.
175	0307	1		! (SUCC, PRED)
176	0308	1	CASING_SIGNIFICANT,	! Set to TRUE if upper/lower casing
177	0309	1		! is significant in names.
178	0310	1	CHARPTR: REF VECTOR[.BYTE],	! Pointer to current character in input
179	0311	1		! line being scanned and parsed
180	0312	1	CHARTBL: CHRTBL\$TABLE,	! Character Table for current language
181	0313	1	COMPONENTS_IN_PATHNAME,	! Set to TRUE if the current language
182	0314	1		! picks up all record components
183	0315	1		! before calling GETSYMBOL.
184	0316	1	ENFORCE_RECORD,	! Set to TRUE if the current language
185	0317	1		! requires that COMP be a
186	0318	1		! component of REC in the
187	0319	1		! expression REC.COMP
188	0320	1	EXPRESSION_RADIX,	! Radix value to be used to interpret
189	0321	1		! constants in expressions
190	0322	1	IDENT_OPERATOR_TABLE:	! Pointer to an Operator Table for the
191	0323	1	REF VECTOR[.LONG],	! current language where the sym-
192	0324	1		! bols are identifiers (NOT, AND)
193	0325	1	INCOMPLETE_QUAL,	! Set to TRUE if incomplete data
194	0326	1		! qualification is allowed in
195	0327	1		! then current language.
196	0328	1		! (e.g., A.C in place of A.B.C)
197	0329	1	MULTIPLE_SUBSCR,	! Set to TRUE if multiple subscripts
198	0330	1		! parens (X[1,2][3]) are allowed
199	0331	1	OPCHAR_OPERATOR_TABLE:	! Pointer to an Operator Table for the
200	0332	1	REF VECTOR[.LONG],	! current language where the sym-
201	0333	1		! bols are special characters (+, **)
202	0334	1	PRIMARY_TABLE: REF PRIMARY\$TABLE,	! Pointer to the Primary Parser State
203	0335	1		! Table for the current language


```

: 204      0336 1  SAVED_TOKEN: INITIAL(0),
: 205      0337 1
: 206      0338 1  STATE_TABLE: REF NUMST$TABLE,
: 207      0339 1
: 208      0340 1  SUBSCRIPT_TERM_TBL:
: 209      0341 1      REF VECTOR[,LONG],
: 210      0342 1  PRIDTBL: REF VECTOR[,LONG],
: 211      0343 1  TERMINATOR_CODE:
: 212      0344 1      INITIAL(TOKEN$K_TERM_NONE),
: 213      0345 1  TERMINATOR_LENGTH: INITIAL(0),
: 214      0346 1
: 215      0347 1  VARSTACK1: VECTOR[VARSTK_SIZE],
: 216      0348 1  VARSTACK2: VECTOR[VARSTK_SIZE],
: 217      0349 1  VARSTACK3: VECTOR[VARSTK_SIZE],
: 218      0350 1  VARSTK_INDEX;

```

```

! Token pointer saved by Primary Parser
!   until next call by Expr. Parser
! Pointer to the Number Scanner State
!   Table for the current language
! Pointer to Terminator Table to use
!   when picking up subscripts
! Pointer to Predefined Identifier Table
! Terminator code for last terminator
!   token found by lexical scanner
! Length in characters of last termina-
!   tor found by the lexical scanner
! Stack of Variant Set RST Entry pointers
! Stack of RST Variant Entry pointers
! Stack of variant component indices
! Current index for VARSTACKn stacks

```


MACROS TO GENERATE PARSE TABLES

These macros are used to generate the parse tables used by the Lexical Scanner and the Parser to parse both language-independent and language-specific constructs accepted by DEBUG.

CHARACTER TABLE

Define the macros which generate the Character table. This includes the macros which generate the base Character Table for language UNKNOWN and the macros which generate the Character Exception Tables for the individual languages. The Character Exception Table for a language lists those characters which have different characteristics in that language that the default characteristics specified for language UNKNOWN. The Character Table for a language is thus formed by copying the default table for language UNKNOWN and then overlaying the entries for those characters listed in the language's Character Exception Table.

These macros are used as follows. To generate the default Character Table, the following sequence of macro invocations is used:

```
CHAR_TABLE(TBLNAME,  
           CHAR_ENT(CHAR, CLASS, BIT1, BIT2, ...)  
           CHAR_ENT(CHAR, CLASS, BIT1, BIT2, ...));
```

Here TBLNAME is the name of the table to be built. The CHAR_TABLE macro defines the whole Character Table as a BLOCKVECTOR and causes each element in the vector to be filled with zeroes unless this is overridden by an explicit CHAR_ENT invocation for that specific character.

The CHAR_ENT macro sets the character characteristics for a specified character. Here CHAR is the character itself (e.g., 'A'), CLASS is the Number Scanner Character Class, and BIT1, BIT2, ... are the names of the other characteristics bits to be set for this character. The CLASS parameter is automatically prefixed by 'NUMST&K_CLASS_' by the macro. Also, the BITn parameters are automatically prefixed by 'CHRTBL&M_' to generate the proper mask value names.

To generate a Character Exception Table for a language, the following macro invocations are used:

```
CHAR_EXCEPTION_TABLE(TBLNAME,  
                     CHAR_ENTRY(CHAR, CLASS, BIT1, BIT2, ...)  
                     CHAR_ENTRY(CHAR, CLASS, BIT1, BIT2, ...));
```

Here CHAR, CLASS, and BIT1, BIT2, ... have the same meanings as for the CHAR_ENT macro above. Similarly, TBLNAME is the name given to the new Character Exception Table. The only difference between these macros and those described above is that a different data structure is generated.

```
277      0408 1 MACRO
278 M 0409 1   CHAR_TABLE(TBLNAME) =
279      0410 1       OWN TBLNAME: VECTOR[256,LONG] PSECT(DBG$PLIT) PRESET(%REMAINING) %;
280      0411 1
281      0412 1 MACRO
282 M 0413 1   CHAR_ENT(CHAR, CLASS) =
283      0414 1       [CHAR]=(((%NAME('NUMST$K_CLASS_',CLASS)^4) OR CHAR_FLAGS(%REMAINING)) %;
284      0415 1
285      0416 1 MACRO
286 M 0417 1   CHAR_FLAGS[FLAG] =
287      0418 1       %NAME('CHRTBL$M_',FLAG) %;
288      0419 1
289      0420 1 MACRO
290 M 0421 1   CHAR_EXCEPTION_TABLE(TBLNAME) =
291 M 0422 1       %IF %LENGTH EQL 1
292 M 0423 1       %THEN
293 M 0424 1           BIND TBLNAME = PLIT(REP 0 OF (0)): VECTOR[,LONG]
294 M 0425 1       %ELSE
295 M 0426 1           BIND TBLNAME = PLIT(%REMAINING): VECTOR[,LONG]
296      0427 1       %FI %;
297      0428 1
298      0429 1 MACRO
299 M 0430 1   CHAR_ENTRY(CHAR, CLASS) =
300      0431 1       CHAR^24 OR %NAME('NUMST$K_CLASS_',CLASS)^4 OR CHAR_FLAGS(%REMAINING) %;
301      0432 1
302      0433 1
303      0434 1 ! Define fields of Character Exception Table entry. These definitions are only
304      0435 1 ! used in the DBG$PARSER_SET_LANGUAGE routine below.
305      0436 1
306      0437 1 FIELD CE_FLDS =
307      0438 1     SET
308      0439 1     CE_BITS = [ 0, V(0,24) ],      ! Character characteristics bits
309      0440 1     CE_CHAR = [ 0, B3_ ]          ! The character itself
310      0441 1     TES;
311      0442 1
312      0443 1 MACRO
313      0444 1     CE_ENTRY = BLOCK[1,WORD] FIELD(CE_FLDS) %;
314      0445 1
315      0446 1
316      0447 1
317      0448 1 ! OPERATOR TABLE
318      0449 1
319      0450 1
320      0451 1 ! Define the macros which generate an Operator Table for a language. An Opera-
321      0452 1 ! tor Table is a counted vector (a PLIT) of longwords which point to Operator
322      0453 1 ! Lexical Token Entries for the operators of the language. Each pointer is
323      0454 1 ! relative to the address TABLEBASE so that the code is completely Position-
324      0455 1 ! Independent (PIC); the true pointer value is thus the longword value plus
325      0456 1 ! TABLEBASE.
326      0457 1
327      0458 1 ! An Operator Table is generated with the following macro invocations:
328      0459 1
329      0460 1     OPERATOR_TABLE(TBLNAME,
330      0461 1       OPERATOR_ENTRY(OPNAME, CODE, KIND, LEFT_PREC, RIGHT_PREC, FLAG1, ...),
331      0462 1       OPERATOR_ENTRY(OPNAME, CODE, KIND, LEFT_PREC, RIGHT_PREC, FLAG1, ...));
332      0463 1
333      0464 1
```



```
334 0465 1 Here TBLNAME is the name of the Operator Table generated. OPNAME is the
335 0466 1 quoted character string which constitutes the operator (e.g., '+', ':=',
336 0467 1 'AND', 'EQ.'), CODE is the name of the operation to be performed (e.g.,
337 0468 1 SUBSCRIPT, ADD, OPENPAREN--this is automatically prefixed by 'TOKEN$K',
338 0469 1 by the macro), KIND is the operator kind (PREFIX, INFIX, or POSTFIX),
339 0470 1 LEFT_PREC is its left precedence, and RIGHT_PREC is its right precedence.
340 0471 1 FLAGT, ... is zero or more optional parameters which specify flag bits
341 0472 1 to be set in the Token Entry. Each FLAGn name is automatically prefixed
342 0473 1 by 'TOKEN$M' by the macro to generate the appropriate mask value. The
343 0474 1 only flag at present is PRIMARY which means that the operator is an
344 0475 1 operator within a Primary Symbol.
345 0476 1
346 0477 1 The OPERATOR_ENTRY macro can also be used independently to generate an
347 0478 1 Operator Lexical Token Entry. It returns the address of the Operator
348 0479 1 Lexical Token Entry it created.
349 0480 1
350 0481 1 MACRO
351 M 0482 1 OPERATOR_TABLE(TBLNAME) =
352 M 0483 1     %IF %LENGTH EQL 1
353 M 0484 1     %THEN
354 M 0485 1         BIND TBLNAME = PLIT(REP 0 OF (0)): VECTOR[,LONG]
355 M 0486 1     %ELSE
356 M 0487 1         BIND TBLNAME = PLIT(OP_ENT(%REMAINING)): VECTOR[,LONG]
357 M 0488 1     %FI %;
358 0489 1
359 0490 1 MACRO
360 M 0491 1 OP_ENT[ADDRESS] =
361 0492 1     (ADDRESS) - TABLEBASE %;
362 0493 1
363 0494 1 MACRO
364 M 0495 1 OPERATOR_ENTRY(OPNAME, CODE, KIND, LEFT_PREC, RIGHT_PREC) =
365 M 0496 1     UPLIT BYTE(%NAME('TOKEN$K_', KIND, '_OP'),
366 M 0497 1         %IF %LENGTH LEQ 5
367 M 0498 1         %THEN
368 M 0499 1             0
369 M 0500 1         %ELSE
370 M 0501 1             0 OR OP_FLAGS(%REMAINING)
371 M 0502 1         %FI,
372 M 0503 1         WORD (%NAME('TOKEN$K_', CODE)),
373 M 0504 1         RIGHT_PREC, LEFT_PREC, 0, 0, 0, 0, 0, 0,
374 M 0505 1         %ASCIZ OPNAME) %;
375 0506 1
376 0507 1 MACRO
377 M 0508 1 OP_FLAGS[FLAGNAME] =
378 0509 1     %NAME('TOKEN$M_', FLAGNAME) %;
379 0510 1
380 0511 1
381 0512 1
382 0513 1 BUILT-IN-FUNCTION TABLE
383 0514 1
384 0515 1
385 0516 1 Define the macros which generate an Built-in Function Table for a language.
386 0517 1 A Built-in Function Table is a counted vector (a PLIT) of longwords which
387 0518 1 point to Operand Lexical Token Entries for the Built-in Functions of the
388 0519 1 language. Each pointer is relative to the address TABLEBASE so that the
389 0520 1 code is completely Position-Independent (PIC); the true pointer value is
390 0521 1 thus the longword value plus TABLEBASE.
```



```
391 0522 1
392 0523 1
393 0524 1
394 0525 1
395 0526 1
396 0527 1
397 0528 1
398 0529 1
399 0530 1
400 0531 1
401 0532 1
402 0533 1
403 0534 1
404 0535 1
405 0536 1
406 0537 1
407 0538 1
408 0539 1
409 0540 1
410 0541 1
411 M 0542 1
412 M 0543 1
413 M 0544 1
414 M 0545 1
415 M 0546 1
416 M 0547 1
417 0548 1
418 0549 1
419 0550 1
420 M 0551 1
421 0552 1
422 0553 1
423 0554 1
424 M 0555 1
425 M 0556 1
426 M 0557 1
427 M 0558 1
428 M 0559 1
429 0560 1
430 0561 1
431 0562 1
432 0563 1
433 0564 1
434 0565 1
435 0566 1
436 0567 1
437 0568 1
438 0569 1
439 0570 1
440 0571 1
441 0572 1
442 0573 1
443 0574 1
444 0575 1
445 0576 1
446 0577 1
447 0578 1

An Built-in Function Table is generated with the following macro invocations:

    BUILT_IN_FUNCTION_TABLE(TBLNAME,
        BUILT_IN_FUNCTION_ENTRY(OPNAME, CODE),
        BUILT_IN_FUNCTION_ENTRY(OPNAME, CODE);

Here TBLNAME is the name of the Built-in Function Table generated. OPNAME
is the quoted character string which constitutes the Built-in Function call
(e.g., 'succ', 'pred'), CODE is the name of the Built-in Function to be
performed (e.g., SUCC, PRED--this is automatically prefixed by 'TOKEN$K'
by the macro), and ARGUMENTS is the number of arguments for the Built-in
function.

The BUILT_IN_FUNCTION_ENTRY macro can also be used independently to
generate an Built-in Function Lexical Token Entry. It returns the
address of the Built-in Function Lexical Token Entry it created.

MACRO
    BUILT_IN_FUNCTION_TABLE(TBLNAME) =
        %IF %LENGTH EQL 1
            %THEN
                BIND TBLNAME = PLIT(REP 0 OF (0)): VECTOR[,LONG]
            %ELSE
                BIND TBLNAME = PLIT(BIF_ENT(%REMAINING)): VECTOR[,LONG]
            %FI %;

MACRO
    BIF_ENT[ADDRESS] =
        -(ADDRESS) - TABLEBASE %;

MACRO
    BUILT_IN_FUNCTION_ENTRY(OPNAME, CODE, ARGUMENTS) =
        UPLIT BYTE(TOKEN$K OPERAND, ARGUMENTS,
            WORD(TOKEN$K BUILT_IN_FUNCTION),
            BYTE(%NAME('TOKEN$K_', CODE)), 0,
            0, 0,
            %ASCII OPNAME) %;

!! Define two tokens that are used in other modules.
GLOBAL BIND
    DBG$GL_CONVERT_TOKEN =
        OPERATOR_ENTRY ('CONVERT', CONVERT, INFIX, 0, 0),
    DBG$GL_DEPOSIT_TOKEN =
        OPERATOR_ENTRY ('DEPOSIT', DEPOSIT, INFIX, 0, 0),
    DBG$GL_IDENTITY_TOKEN =
        OPERATOR_ENTRY ('IDENTITY', IDENTITY, PREFIX, 0, 0),
    DBG$GL_NEG_CONST_TOKEN =
        OPERATOR_ENTRY ('NEGCONST', NEGCONST, PREFIX, 0, 0),
    DBG$GL_POS_CONST_TOKEN =
        OPERATOR_ENTRY ('POSCONST', POSCONST, PREFIX, 0, 0),
    DBG$GL_NEG_SIGN_TOKEN =
        OPERATOR_ENTRY ('-', UNARY_MINUS, PREFIX, 0, 0),
    DBG$GL_POS_SIGN_TOKEN =
```

```
448 0579 1 OPERATOR_ENTRY ('+', UNARY_PLUS, PREFIX, 0, 0);
449 0580 1
450 0581 1
451 0582 1 OPERAND LEXICAL TOKEN ENTRIES
452 0583 1
453 0584 1
454 0585 1 Define macro which builds an Operand Lexical Token Entry and returns the
455 0586 1 address of the entry. The macro is used like this:
456 0587 1
457 0588 1 OPERAND_ENTRY(TOKENCODE, NAMESTRING)
458 0589 1
459 0590 1 Here TOKENCODE is the code value for the type of operand this token
460 0591 1 constitutes. TOKEN$K_IDENTIFIER or TOKEN$K_INTEGER are valid examples.
461 0592 1 NAMESTRING is the ASCII name to be associated with the operand token.
462 0593 1
463 0594 1 MACRO
464 0595 1 OPERAND_ENTRY(CODE, NAMESTRING) =
465 0596 1 UPLIT BYTE(TOKEN$K_OPERAND, 0, WORD (CODE),
466 0597 1 0, 0, 0, 0,
467 0598 1 %ASCII(NAMESTRING): TOKEN$ENTRY %;
468 0599 1
469 0600 1
470 0601 1
471 0602 1 PREDEFINED IDENTIFIER TABLE
472 0603 1
473 0604 1
474 0605 1 Define the macros which generate a Predefined Identifier Table for a language.
475 0606 1 A Predefined Identifier Table is a counted vector (a PLIT) of longwords which
476 0607 1 point to Predefined Identifier Entries for the reserved names of the language.
477 0608 1 Each pointer is relative to the address TABLEBASE so that the code is completely
478 0609 1 Position Independent (PIC); the true pointer value is thus the longword value
479 0610 1 plus TABLEBASE.
480 0611 1
481 0612 1 A Predefined Identifier Table is generated with the following macro invocations:
482 0613 1
483 0614 1 PRID_TABLE(TBLNAME,
484 0615 1 PRID_ENTRY(PRIDNAME, FCODE, DTYPE, VALUE),
485 0616 1 PRID_ENTRY(PRIDNAME, FCODE, DTYPE, VALUE));
486 0617 1
487 0618 1
488 0619 1 Here TBLNAME is the name of the Predefined ID Table generated. PRIDNAME is
489 0620 1 the quoted character string which constitutes the reserved name (e.g.,
490 0621 1 'TRUE', 'FALSE', '.TRUE.', '.FALSE.'). DTYPE is the Data type (e.g., Boolean,
491 0622 1 Integer). DTYPE is automatically prefixed by 'DSC$K_DTYPE', and VALUE is a
492 0623 1 longword constant value for the PRID constant, FCODE is the format Code.
493 0624 1
494 0625 1 The PRID_ENTRY macro can also be used independently to generate an entry.
495 0626 1 It returns the address of the PRID Entry it created.
496 0627 1
497 0628 1 MACRO
498 0629 1 PRID_TABLE(TBLNAME) =
499 0630 1 %IF %LENGTH EQL 1
500 0631 1 %THEN
501 0632 1 BIND TBLNAME = PLIT(REP 0 OF (0)): VECTOR[,LONG]
502 0633 1 %ELSE
503 0634 1 BIND TBLNAME = PLIT(PRID_ENT(%REMAINING)): VECTOR[,LONG]
504 0635 1 %FI %;
```

```
505 0636 1
506 0637 1 MACRO
507 M 0638 1 PRID_ENT[ADDRESS] =
508 0639 1 (ADDRESS) - TABLEBASE %;
509 0640 1
510 0641 1 MACRO
511 M 0642 1 PRID_ENTRY(NAMESTRING, FCODE, DTYPE, VALUE) =
512 M 0643 1 OPLIT BYTE(PRID$K_CONSTANT,
513 M 0644 1 %NAME('DSC$K_DTYPE_', DTYPE),
514 M 0645 1 %NAME('RST$K_TYPE_', FCODE),
515 M 0646 1 0,
516 M 0647 1 LONG (VALUE),
517 0648 1 %ASCII NAMESTRING)%;
518 0649 1
519 0650 1
520 0651 1
521 0652 1 TERMINATOR LEXICAL TOKEN ENTRIES
522 0653 1
523 0654 1
524 0655 1 Define the macros which generate Terminator Lexical Token Entries. These
525 0656 1 entries define lexical tokens which can terminate the current expression.
526 0657 1 For example, subscript expressions can normally be terminated by the tokens
527 0658 1 "" and "". These are thus the Terminator Tokens for subscript expressions.
528 0659 1 Similarly, "DO" is a terminator token for the address expression in the
529 0660 1 SET BREAK command and "=" is a terminator token for the address expression
530 0661 1 in the DEPOSIT command. Terminator Lexical Token Entries thus define the
531 0662 1 tokens which can validly terminate the current expression in the current
532 0663 1 context. Terminator Lexical Token Entries have exactly the same format as
533 0664 1 Operand Lexical Token Entries and are referenced with the same field names.
534 0665 1
535 0666 1 A table of Terminator Lexical Token Entries is declared as follows:
536 0667 1
537 0668 1 TERMINATOR TABLE(TBLNAME,
538 0669 1 TERMINATOR_ENTRY(NAMESTRING, CODE, FLAG1, ...),
539 0670 1
540 0671 1 TERMINATOR_ENTRY(NAMESTRING, CODE, FLAG1, ...));
541 0672 1
542 0673 1 Here TBLNAME is the name of the terminator table. NAMESTRING is the ASCII
543 0674 1 string which constitutes the terminator token, CODE is the name of a termi-
544 0675 1 nator token code which identifies what kind of terminator this is (this name
545 0676 1 is automatically prefixed by "TOKEN$K_" by the macro), and FLAG1, ... is
546 0677 1 zero or more flag names indicating flag bits to be set in the Terminator
547 0678 1 Token Entry. The flags names are automatically prefixed by "TOKEN$M_" to
548 0679 1 generate the appropriate mask value name.
549 0680 1
550 0681 1 MACRO
551 M 0682 1 TERMINATOR TABLE(TBLNAME) =
552 M 0683 1 %IF %LENGTH EQL 1
553 M 0684 1 %THEN
554 M 0685 1 BIND TBLNAME = PLIT(REP 0 OF (0)): VECTOR[,LONG]
555 M 0686 1 %ELSE
556 M 0687 1 BIND TBLNAME = PLIT(OP_ENT(%REMAINING)): VECTOR[,LONG]
557 0688 1 %FI %;
558 0689 1
559 0690 1 MACRO
560 M 0691 1 TERMINATOR_ENTRY(NAMESTRING, CODE) =
561 M 0692 1 UPLIT BYTE(0,
```



```

562      %IF %LENGTH LEQ 2
563      %THEN
564      0
565      %ELSE
566      0 OR OP_FLAGS(%REMAINING)
567      %FI,
568      WORD (%NAME('TOKENSK_',CODE)),
569      0, 0, 0, 0
570      %ASCIC NAMESTRING) %;
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
```

NUMBER SCANNER STATE TABLE

Define the macros which generate Number Scanner State Tables. These tables are used in the lexical scanning of numeric constants and are in general specific to each language.

A Number Scanner State Table for a language is generated with the following macro invocations:

```

        NUMBER_STATE_TABLE(TBLNAME,
            NUMBER_STATE(STATE_ID,
                NUMBER_TRANSITION(CHAR_CLASS, ACTION, NEXTSTATE),
                NUMBER_TRANSITION(CHAR_CLASS, ACTION, NEXTSTATE),
                NUMBER_TRANSITION(OTHER, ACTION, NEXTSTATE)),
            ...
            NUMBER_STATE(STATE_ID,
                NUMBER_TRANSITION(CHAR_CLASS, ACTION, NEXTSTATE),
                NUMBER_TRANSITION(CHAR_CLASS, ACTION, NEXTSTATE),
                NUMBER_TRANSITION(OTHER, ACTION, NEXTSTATE)));
```

The NUMBER_STATE_TABLE macro sets up the Number Scanner State Table as a whole and binds the table name TBLNAME to that structure. Each state in the table is declared with the NUMBER_STATE macro whose STATE_ID argument names the state. The actual state name is prefixed by 'NUMST\$XX_STATE_' by the macro and must be declared as such in the COMPILETIME declaration below.

The NUMBER_TRANSITION macro defines one transition from the current state to some other state. The transition is taken if the next input character is of the character class specified by CHAR_CLASS. (CHAR_CLASS is automatically prefixed by 'NUMST\$K_CLASS_' to generate the class code constant.) If the transition is taken, the action specified by ACTION is taken. ACTION, which is automatically prefixed by 'NUMST\$K_ACT_' by the macro, is a CASE index used by the Number Scanner to select an action routine which performs whatever semantic processing is appropriate. After the action routine executes, the next state in the state table is given by NEXTSTATE. NEXTSTATE is automatically prefixed by 'NUMST\$K_STATE_' by the macro and must be declared as the STATE_ID on some other NUMBER_STATE declaration in the state table.

Every state in the state table must have at least two transitions and the

```
619 0750 1 : Last transition must always be for class NUMST$K CLASS OTHER, written as
620 0751 1 : OTHER in the NUMBER_TRANSITION invocation. (There is no point to having
621 0752 1 : a state with only a single transition since that state could then be merged
622 0753 1 : with the transition's next state.)
623 0754 1 :
624 0755 1 MACRO
625 M 0756 1 NUMBER_STATE_TABLE(TBLNAME) =
626 M 0757 1 BIND TBLNAME = UPLIT(%REMAINING): NUMST$TABLE
627 0758 1 %ASSIGN(NUMST$XX_CUR_LOC, 0) %;
628 0759 1
629 0760 1 MACRO
630 M 0761 1 NUMBER_STATE(STATE_ID) =
631 M 0762 1 %ASSIGN(%NAME('NUMST$XX STATE ', STATE_ID), NUMST$XX_CUR_LOC)
632 M 0763 1 %ASSIGN(NUMST$XX_CUR_LOC, NUMST$XX_CUR_LOC + (%LENGTH - 1)/3)
633 0764 1 %REMAINING %;
634 0765 1
635 0766 1 MACRO
636 M 0767 1 NUMBER_TRANSITION(CHAR CLASS, ACTION, NEXTSTATE) =
637 M 0768 1 BYTE (%NAME('NUMST$K CLASS ', CHAR CLASS)),
638 M 0769 1 BYTE (%NAME('NUMST$K ACT ', ACTION)),
639 0770 1 WORD (%NAME('NUMST$XX STATE ', NEXTSTATE)) %;
640 0771 1
641 0772 1 COMPILETIME
642 0773 1 NUMST$XX_CUR_LOC = 0, : Current index into state table vector
643 0774 1 : during macro expansion
644 0775 1 NUMST$XX_STATE_START_STATE = 0, : Index values in state table vector
645 0776 1 NUMST$XX_STATE_LEADING_DOT = 0, : for state table entries
646 0777 1 NUMST$XX_STATE_ACCUM_INT = 0,
647 0778 1 NUMST$XX_STATE_T_ACCUM_INT = 0,
648 0779 1 NUMST$XX_STATE_ACCUM_HEX = 0,
649 0780 1 NUMST$XX_STATE_ACCUM_FRAC = 0,
650 0781 1 NUMST$XX_STATE_T_ACCUM_FRAC = 0,
651 0782 1 NUMST$XX_STATE_GET_EXPONENT = 0,
652 0783 1 NUMST$XX_STATE_GET_EXP_SIGN = 0,
653 0784 1 NUMST$XX_STATE_ACCUM_EXP = 0,
654 0785 1 NUMST$XX_STATE_T_ACCUM_EXP = 0,
655 0786 1 NUMST$XX_STATE_B_START_STATE = 0,
656 0787 1 NUMST$XX_STATE_B_ACCUM_INT = 0,
657 0788 1 NUMST$XX_STATE_T_B_ACCUM_INT = 0,
658 0789 1 NUMST$XX_STATE_B_ACCUM_FRAC = 0,
659 0790 1 NUMST$XX_STATE_T_B_ACCUM_FRAC = 0,
660 0791 1 NUMST$XX_STATE_END_STATE = 0;
661 0792 1
662 0793 1
663 0794 1 : PRIMARY PARSER STATE TABLE
664 0795 1 :
665 0796 1 :
666 0797 1 : Define the macros which generate Primary Parser State Tables. These tables
667 0798 1 : are used in the parsing of Primary Symbols (symbol names including pathname
668 0799 1 : qualification, data qualification, subscripting, dereferencing, etc.) and
669 0800 1 : are specific to each language.
670 0801 1 :
671 0802 1 : A Primary Parser State Table for a language is generated with the following
672 0803 1 : macro invocations:
673 0804 1 :
674 0805 1 : PRIMARY_STATE_TABLE(TBLNAME,
675 0806 1 :
```

```
676 0807 1 PRIMARY STATE(STATE ID,  
677 0808 1 PRIMARY_TRANSITION(OPCODE, ACTION, NEXTSTATE),  
678 0809 1  
679 0810 1 PRIMARY_TRANSITION(OPCODE, ACTION, NEXTSTATE)),  
680 0811 1  
681 0812 1 ...  
682 0813 1  
683 0814 1 PRIMARY STATE(STATE ID,  
684 0815 1 PRIMARY_TRANSITION(OPCODE, ACTION, NEXTSTATE),  
685 0816 1  
686 0817 1 PRIMARY_TRANSITION(OPCODE, ACTION, NEXTSTATE)));  
687 0818 1  
688 0819 1 The PRIMARY STATE TABLE macro sets up the Primary Parser State Table as a  
689 0820 1 whole and binds the table name TBLNAME to that data structure. Each state in  
690 0821 1 the table is declared with the PRIMARY_STATE macro whose STATE_ID argument  
691 0822 1 names the state. The actual state name is prefixed by 'PRIMARY$XX STATE_'  
692 0823 1 by the macro and must be declared in the COMPILETIME declaration below.  
693 0824 1  
694 0825 1 The PRIMARY_TRANSITION macro defines one transition from the current state  
695 0826 1 to some other state. The transition is taken if the current Primary Operator  
696 0827 1 returned by the Lexical Scanner has the Operator Code (TOKEN$W_CODE) speci-  
697 0828 1 fied as OPCODE. (OPCODE is automatically prefixed by 'TOKEN$K_' to generate  
698 0829 1 the constant name.) If the transition is taken, the action routine specified  
699 0830 1 by ACTION is taken. ACTION, which the macro prefixes with 'PRIMARY$K_ACT_',  
700 0831 1 is a CASE index used by the Primary Parser to select an action routine to do  
701 0832 1 whatever semantic processing is appropriate to build the Primary Descriptor  
702 0833 1 for the Primary Symbol being parsed. After the action routine executes, the  
703 0834 1 next state in the state table is given by NEXTSTATE. NEXTSTATE is automati-  
704 0835 1 cally prefixed by 'PRIMARY$XX STATE_' by the macro and must be declared as  
705 0836 1 the STATE_ID on some other PRIMARY_STATE declaration in the state table.  
706 0837 1  
707 0838 1 If a Primary Operator is encountered during the scan for which there is no  
708 0839 1 transition in the current Primary Parser State Table state, the Primary  
709 0840 1 Parser signals an error. This is how many ill-formed Primary Symbols are  
710 0841 1 detected. Every state thus has an implicit transition for every unspecified  
711 0842 1 Operator Code (the "other" transition) for which the action is to signal a  
712 0843 1 syntax error.  
713 0844 1  
714 0845 1 MACRO  
715 M 0846 1 PRIMARY STATE TABLE(TBLNAME) =  
716 M 0847 1 BIND TBLNAME = UPLIT(%REMAINING): PRIMARY$TABLE  
717 0848 1 %ASSIGN(PRIMARY$XX_CUR_LOC, 0) %;  
718 0849 1  
719 0850 1 MACRO  
720 M 0851 1 PRIMARY STATE(STATE ID) =  
721 M 0852 1 %ASSIGN(%NAME('PRIMARY$XX STATE ', STATE ID), PRIMARY$XX_CUR_LOC)  
722 M 0853 1 %ASSIGN(PRIMARY$XX_CUR_LOC, PRIMARY$XX_CUR_LOC + (%LENGTH - T)/3 + 1)  
723 0854 1 %IF %LENGTH GTR 1 %THEN %REMAINING, %FT LONG (0) %;  
724 0855 1  
725 0856 1 MACRO  
726 M 0857 1 PRIMARY_TRANSITION(OPCODE, ACTION, NEXTSTATE) =  
727 M 0858 1 BYTE (%NAME('TOKEN$K_', OPCODE))  
728 M 0859 1 BYTE (%NAME('PRIMARY$K_ACT ', ACTION))  
729 0860 1 WORD (%NAME('PRIMARY$XX_STATE_', NEXTSTATE)) %;  
730 0861 1  
731 0862 1 COMPILETIME  
732 0863 1 PRIMARY$XX_CUR_LOC = 0, ! Current index into primary parser state
```



```
733 0864 1
734 0865 1 PRIMARY$XX_STATE_START_STATE = 0,
735 0866 1 PRIMARY$XX_STATE_GET_GLOBAL = 0,
736 0867 1 PRIMARY$XX_STATE_GOT_BACKSLASH = 0,
737 0868 1 PRIMARY$XX_STATE_GOT_DOT = 0,
738 0869 1 PRIMARY$XX_STATE_GOT_SUBSCRIPT = 0,
739 0870 1 PRIMARY$XX_STATE_GOT_SUBSCRIPT2 = 0,
740 0871 1 PRIMARY$XX_STATE_GOT_BRACKET = 0,
741 0872 1 PRIMARY$XX_STATE_GOT_DEREF = 0,
742 0873 1 PRIMARY$XX_STATE_GOT_DOT_SLASH = 0,
743 0874 1 PRIMARY$XX_STATE_END_STATE = 0;
744 0875 1
745 0876 1
746 0877 1 ! The following data structure is used in the SAVE_SUBSCRIPTS routine
747 0878 1 ! and the PATHNAME_TO_PRIMARY routine.
748 0879 1
749 0880 1 FIELD
750 0881 1 SUBSCR_DESC_FIELDS =
751 0882 1 SET
752 0883 1 SUBSCR$B_PATH_INDEX = [0,0,8,0],
753 0884 1
754 0885 1
755 0886 1 SUBSCR$B_SUBCNT = [0,8,8,0],
756 0887 1 SUBSCR$V_RANGE = [0,16,1,0],
757 0888 1 SUBSCR$V_aster = [0,17,1,0],
758 0889 1 SUBSCR$V_MARKER = [0,18,1,0],
759 0890 1 SUBSCR$L_LBOUND = [1,0,32,0],
760 0891 1 SUBSCR$L_UBOUND = [2,0,32,0]
761 0892 1 TES;
762 0893 1
763 M 0894 1 MACRO SUBSCR$DESC = BLOCKVECTOR(DBG$K_PATHNAME_SIZE,3, LONG]
764 0895 1 FIELD(SUBSCR_DESC_FIELDS)%;
765 0896 1
766 0897 1 LITERAL SUBSCR_DESC_SIZE = 3*DBG$K_PATHNAME_SIZE*XUPVAL;
767 0898 1
768 0899 1
769 0900 1 !
770 0901 1 !
771 0902 1 !
772 0903 1 ! Define parser flags which may have different settings for different languages.
773 0904 1 ! These flags are stored in the Table of Language-Specific Tables and are copied
774 0905 1 ! to individual OWN variables when the current language is set. These flags
775 0906 1 ! control the behavior of the parser in those cases where different languages
776 0907 1 ! have different behaviors and the easiest way of encoding these differences
777 0908 1 ! is to have a flags to control the parser. The following flags are available:
778 0909 1
779 0910 1 !
780 0911 1 !
781 0912 1 !
782 0913 1 !
783 0914 1 !
784 0915 1 !
785 0916 1 !
786 0917 1 !
787 0918 1 !
788 0919 1 !
789 0920 1 !
```

table during macro expansion
Index values in primary parser state
table vector for the states

Says which pathname element
the subscripts were
associated with
Count of subscripts so far
True if range was specified
True for '*' range
Marker bit (see SAVE_SUBSCRIPTS)
Lower bound of range
Upper bound of range

MULTIPLE_SUBSCR - The language allows multiple subscript parentheses
in array references. For example, PASCAL allows
X[1,2,3,4] to be written as X[1,2][3][4]. This
flag should not be set if the second form is not
equivalent to the first form of array reference.

Each of these flags is set to TRUE or FALSE in an OWN variable with the same
name as the flag name give here.

```
790 0921 1 | TABLE OF LANGUAGE-SPECIFIC TABLES
791 0922 1 |
792 0923 1 |
793 0924 1 | Define the macro which builds the table of pointers to the language-specific
794 0925 1 | parse tables. This table is simply a vector of pointers to the Character
795 0926 1 | Exception Table, the Identifier Operator table, the Operator Character Opera-
796 0927 1 | tor Table, and the Number Scanner State Table for the specified language.
797 0928 1 | Each pointer is relative to TABLEBASE to keep the table position-independent
798 0929 1 | (PIC). The macro is used as follows:
799 0930 1 |
800 0931 1 |     LANGUAGE_TABLES(LANGUAGE = language-name,
801 0932 1 |                     CHARTBL = character-exception-table,
802 0933 1 |                     IDENT_OPTBL = identifier-operator-table,
803 0934 1 |                     OPCHAR_OPTBL = operator-character-operator-table,
804 0935 1 |                     NUMBER_TABLE = number-scanner-state-table,
805 0936 1 |                     PRIMARY_TABLE = primary-parser-state-table,
806 0937 1 |                     SUBSCR_TERMS = subscript-terminator-table,
807 0938 1 |                     PRIDTBL = Predefined-identifier-table,
808 0939 1 |                     BIF_TABLE = built-in-function-table,
809 0940 1 |                     MULTIPLE SUBSCR = true-or-false,
810 0941 1 |                     ENFORCE_RECORD = true-or-false,
811 0942 1 |                     CASING_SIGNIFICANT = true-or-false,
812 0943 1 |                     COMPONENTS_IN_PATHNAME = true-or-false,
813 0944 1 |                     INCOMPLETE_QUAL = true-or-false);
814 0945 1 |
815 0946 1 | Notice that keyword parameters are used. The meanings of the parameters
816 0947 1 | should be self-explanatory.
817 0948 1 |
818 0949 1 | KEYWORDMACRO
819 0950 1 |     LANGUAGE_TABLES(LANGUAGE=DBG$K_UNKNOWN, CHARTBL=0, IDENT_OPTBL=0,
820 0951 1 |                     OPCHAR_OPTBL=0, NUMBER_TABLE=0, PRIMARY_TABLE=0,
821 0952 1 |                     SUBSCR_TERMS=0, PRIDTBL=0, BIF_TABLE=0,
822 0953 1 |                     MULTIPLE SUBSCR=FALSE, ENFORCE_RECORD=TRUE,
823 0954 1 |                     CASING_SIGNIFICANT=FALSE,
824 0955 1 |                     COMPONENTS_IN_PATHNAME=FALSE, INCOMPLETE_QUAL=FALSE) =
825 0956 1 |
826 0957 1 |     BIND %NAME(LANGUAGE, 'TABLES') =
827 0958 1 |         UPLIT (CHARTBL - TABLEBASE,
828 0959 1 |               IDENT_OPTBL - TABLEBASE,
829 0960 1 |               OPCHAR_OPTBL - TABLEBASE,
830 0961 1 |               NUMBER_TABLE - TABLEBASE,
831 0962 1 |               PRIMARY_TABLE - TABLEBASE,
832 0963 1 |               SUBSCR_TERMS - TABLEBASE,
833 0964 1 |               PRIDTBL - TABLEBASE,
834 0965 1 |               BIF_TABLE - TABLEBASE,
835 0966 1 |               MULTIPLE SUBSCR,
836 0967 1 |               ENFORCE_RECORD,
837 0968 1 |               CASING_SIGNIFICANT,
838 0969 1 |               COMPONENTS_IN_PATHNAME,
839 0970 1 |               INCOMPLETE_QUAL): VECTOR [,LONG] %;
```

LANGUAGE - INDEPENDENT PARSE TABLES

The Parser and Lexical Scanner tables in this section are the language-independent tables used during lexical scanning and parsing.

Define a table 'base address'. The TABLEBASE label defines a location in the PLIT PSECT which constitutes the base address for all pointers within the DEBUG parse tables defined in this module. This base address is needed to make these tables position-independent (PIC) since DEBUG may be placed anywhere in the virtual address space when run with a user program.

BIND
TABLEBASE = UPLIT BYTE (%ASCII 'BASE');

Generate the 'Percent Table' to define all built-in '%' symbols recognized by DEBUG. This includes %LINE, %LABEL, %NAME, and all the register names.

Define literals which identify the kind of '%' symbol a given such symbol is. This is used as a CASE index for further processing in the Lexical Scanner (DBG\$LEXICAL_SCANNER).

LITERAL

PERCENT_NOFIND	= 0,	No such '%' symbol exists
PERCENT_LINE	= 1,	%LINE symbol
PERCENT_LABEL	= 2,	%LABEL symbol
PERCENT_NAME	= 3,	%NAME symbol
PERCENT_DEC	= 4,	%DEC decimal operator
PERCENT_HEX	= 5,	%HEX hexadecimal operator
PERCENT_OCT	= 6,	%OCT octal operator
PERCENT_BIN	= 7,	%BIN binary operator
PERCENT_IDENT	= 8,	Identifier '%' symbols such as %R5

Generate the actual Percent Table itself. Note that for compatibility with past usage, we allow abbreviations for %LINE and %LABEL, but only for those two reserved names.

BIND
PERCENT_TABLE = PLIT(
UPLIT BYTE(PERCENT_LINE, %ASCII '%LINE') - TABLEBASE,
UPLIT BYTE(PERCENT_LINE, %ASCII '%LIN') - TABLEBASE,
UPLIT BYTE(PERCENT_LINE, %ASCII '%LI') - TABLEBASE,
UPLIT BYTE(PERCENT_LABEL, %ASCII '%LABEL') - TABLEBASE,
UPLIT BYTE(PERCENT_LABEL, %ASCII '%LABE') - TABLEBASE,
UPLIT BYTE(PERCENT_LABEL, %ASCII '%LAB') - TABLEBASE,
UPLIT BYTE(PERCENT_LABEL, %ASCII '%LA') - TABLEBASE,
UPLIT BYTE(PERCENT_NAME, %ASCII '%NAME') - TABLEBASE,
UPLIT BYTE(PERCENT_IDENT, %ASCII '%R0') - TABLEBASE,
UPLIT BYTE(PERCENT_IDENT, %ASCII '%R1') - TABLEBASE,
UPLIT BYTE(PERCENT_IDENT, %ASCII '%R2') - TABLEBASE,
UPLIT BYTE(PERCENT_IDENT, %ASCII '%R3') - TABLEBASE,


```
898      1028 1      UPLIT BYTE(PERCENT_IDENT, %ASCIC 'XR4' ) - TABLEBASE,
899      1029 1      UPLIT BYTE(PERCENT_IDENT, %ASCIC 'XR5' ) - TABLEBASE,
900      1030 1      UPLIT BYTE(PERCENT_IDENT, %ASCIC 'XR6' ) - TABLEBASE,
901      1031 1      UPLIT BYTE(PERCENT_IDENT, %ASCIC 'XR7' ) - TABLEBASE,
902      1032 1      UPLIT BYTE(PERCENT_IDENT, %ASCIC 'XR8' ) - TABLEBASE,
903      1033 1      UPLIT BYTE(PERCENT_IDENT, %ASCIC 'XR9' ) - TABLEBASE,
904      1034 1      UPLIT BYTE(PERCENT_IDENT, %ASCIC 'XR10' ) - TABLEBASE,
905      1035 1      UPLIT BYTE(PERCENT_IDENT, %ASCIC 'XR11' ) - TABLEBASE,
906      1036 1      UPLIT BYTE(PERCENT_IDENT, %ASCIC 'XR12' ) - TABLEBASE,
907      1037 1      UPLIT BYTE(PERCENT_IDENT, %ASCIC 'XR13' ) - TABLEBASE,
908      1038 1      UPLIT BYTE(PERCENT_IDENT, %ASCIC 'XR14' ) - TABLEBASE,
909      1039 1      UPLIT BYTE(PERCENT_IDENT, %ASCIC 'XR15' ) - TABLEBASE,
910      1040 1      UPLIT BYTE(PERCENT_IDENT, %ASCIC 'XAP' ) - TABLEBASE,
911      1041 1      UPLIT BYTE(PERCENT_IDENT, %ASCIC 'XFP' ) - TABLEBASE,
912      1042 1      UPLIT BYTE(PERCENT_IDENT, %ASCIC 'XSP' ) - TABLEBASE,
913      1043 1      UPLIT BYTE(PERCENT_IDENT, %ASCIC 'XPC' ) - TABLEBASE,
914      1044 1      UPLIT BYTE(PERCENT_IDENT, %ASCIC 'XPSL' ) - TABLEBASE,
915      1045 1      UPLIT BYTE(PERCENT_IDENT, %ASCIC 'Xr0' ) - TABLEBASE,
916      1046 1      UPLIT BYTE(PERCENT_IDENT, %ASCIC 'Xr1' ) - TABLEBASE,
917      1047 1      UPLIT BYTE(PERCENT_IDENT, %ASCIC 'Xr2' ) - TABLEBASE,
918      1048 1      UPLIT BYTE(PERCENT_IDENT, %ASCIC 'Xr3' ) - TABLEBASE,
919      1049 1      UPLIT BYTE(PERCENT_IDENT, %ASCIC 'Xr4' ) - TABLEBASE,
920      1050 1      UPLIT BYTE(PERCENT_IDENT, %ASCIC 'Xr5' ) - TABLEBASE,
921      1051 1      UPLIT BYTE(PERCENT_IDENT, %ASCIC 'Xr6' ) - TABLEBASE,
922      1052 1      UPLIT BYTE(PERCENT_IDENT, %ASCIC 'Xr7' ) - TABLEBASE,
923      1053 1      UPLIT BYTE(PERCENT_IDENT, %ASCIC 'Xr8' ) - TABLEBASE,
924      1054 1      UPLIT BYTE(PERCENT_IDENT, %ASCIC 'Xr9' ) - TABLEBASE,
925      1055 1      UPLIT BYTE(PERCENT_IDENT, %ASCIC 'Xr10' ) - TABLEBASE,
926      1056 1      UPLIT BYTE(PERCENT_IDENT, %ASCIC 'Xr11' ) - TABLEBASE,
927      1057 1      UPLIT BYTE(PERCENT_IDENT, %ASCIC 'Xr12' ) - TABLEBASE,
928      1058 1      UPLIT BYTE(PERCENT_IDENT, %ASCIC 'Xr13' ) - TABLEBASE,
929      1059 1      UPLIT BYTE(PERCENT_IDENT, %ASCIC 'Xr14' ) - TABLEBASE,
930      1060 1      UPLIT BYTE(PERCENT_IDENT, %ASCIC 'Xr15' ) - TABLEBASE,
931      1061 1      UPLIT BYTE(PERCENT_IDENT, %ASCIC 'Xap' ) - TABLEBASE,
932      1062 1      UPLIT BYTE(PERCENT_IDENT, %ASCIC 'Xfp' ) - TABLEBASE,
933      1063 1      UPLIT BYTE(PERCENT_IDENT, %ASCIC 'Xsp' ) - TABLEBASE,
934      1064 1      UPLIT BYTE(PERCENT_IDENT, %ASCIC 'Xpc' ) - TABLEBASE,
935      1065 1      UPLIT BYTE(PERCENT_IDENT, %ASCIC 'Xpsl' ) - TABLEBASE,
936      1066 1      UPLIT BYTE(PERCENT_IDENT, %ASCIC 'XDEC' ) - TABLEBASE,
937      1067 1      UPLIT BYTE(PERCENT_IDENT, %ASCIC 'XHEX' ) - TABLEBASE,
938      1068 1      UPLIT BYTE(PERCENT_IDENT, %ASCIC 'XOCT' ) - TABLEBASE,
939      1069 1      UPLIT BYTE(PERCENT_IDENT, %ASCIC 'XBIN' ) - TABLEBASE,
940      1070 1      UPLIT BYTE(PERCENT_IDENT, %ASCIC 'XCURLOC' ) - TABLEBASE,
941      1071 1      UPLIT BYTE(PERCENT_IDENT, %ASCIC 'XCURVAL' ) - TABLEBASE,
942      1072 1      UPLIT BYTE(PERCENT_IDENT, %ASCIC 'XPREVLOC' ) - TABLEBASE,
943      1073 1      UPLIT BYTE(PERCENT_IDENT, %ASCIC 'XNEXTLOC' ) - TABLEBASE,
944      1074 1      UPLIT BYTE(PERCENT_IDENT, %ASCIC 'XPARCNT' ) - TABLEBASE,
945      1075 1      ): VECTOR[ , LONG];
946      1076 1
947      1077 1
948      1078 1      ! Define Operator Lexical Token Entries for two predefined operators, namely
949      1079 1      ! the Initiator and the Terminator operators. These operators do not appear
950      1080 1      ! explicitly in the DEBUG command line, but are used internally by the Opera-
951      1081 1      ! tor Precedence Parsers as expression initiator and terminator tokens. The
952      1082 1      ! Primary Terminator is a pseudo-operator used to terminate Primary Symbols.
953      1083 1      !
954      1084 1      BIND
```

```

955 1085 1 INITIATOR_TOKEN =
956 1086 1 OPERATOR_ENTRY('start of expression', INITIATOR, PREFIX, 200, 1),
957 1087 1 TERMINATOR_TOKEN =
958 1088 1 OPERATOR_ENTRY('end of expression', TERMINATOR, POSTFIX, 2, 200, LEXICAL),
959 1089 1 PRIMARY_TERM_TOKEN =
960 1090 1 OPERATOR_ENTRY('end of symbol', PRIMARY_TERM, POSTFIX, 0, 0, PRIMARY);
961 1091 1
962 1092 1
963 1093 1 ! Generate Operator Lexical Token Entries for the radix operators. These
964 1094 1 ! are the operators which change the current expression radix back and forth.
965 1095 1
966 1096 1 BIND
967 1097 1 RADIX_OP_DEC =
968 1098 1 OPERATOR_ENTRY('%DEC', RADIX_DEC, PREFIX, 200, 190, LEXICAL),
969 1099 1 RADIX_OP_HEX =
970 1100 1 OPERATOR_ENTRY('%HEX', RADIX_HEX, PREFIX, 200, 190, LEXICAL),
971 1101 1 RADIX_OP_OCT =
972 1102 1 OPERATOR_ENTRY('%OCT', RADIX_OCT, PREFIX, 200, 190, LEXICAL),
973 1103 1 RADIX_OP_BIN =
974 1104 1 OPERATOR_ENTRY('%BIN', RADIX_BIN, PREFIX, 200, 190, LEXICAL);
975 1105 1
976 1106 1
977 1107 1 ! Generate Operand Lexical Token Entries for the built-in DEBUG symbols
978 1108 1 ! '...', '...', and '...' meaning current location, current value, and previous
979 1109 1 ! location.
980 1110 1
981 1111 1 BIND
982 1112 1 CURLOC_TOKEN = OPERAND_ENTRY(TOKEN$K_IDENTIFIER, '%CURLOC'),
983 1113 1 CURVAL_TOKEN = OPERAND_ENTRY(TOKEN$K_IDENTIFIER, '%CURVAL'),
984 1114 1 PREVLOC_TOKEN = OPERAND_ENTRY(TOKEN$K_IDENTIFIER, '%PREVLOC');
985 1115 1
986 1116 1
987 1117 1 ! Generate the Operator Table for the built-in operators allowed in Address
988 1118 1 ! Expressions.
989 1119 1
990 1120 1 OPERATOR_TABLE(ADDR_EXPR_OPTBL,
991 1121 1 OPERATOR_ENTRY('-', INDIRECT, PREFIX, 200, 40),
992 1122 1 OPERATOR_ENTRY('@', INDIRECT, PREFIX, 200, 40),
993 1123 1 OPERATOR_ENTRY('+', ADD, INFIX, 10, 10),
994 1124 1 OPERATOR_ENTRY('-', SUBTRACT, INFIX, 10, 10),
995 1125 1 OPERATOR_ENTRY('+', UNARY_PLUS, PREFIX, 200, 20),
996 1126 1 OPERATOR_ENTRY('-', UNARY_MINUS, PREFIX, 200, 20),
997 1127 1 OPERATOR_ENTRY('*', MULTIPLY, INFIX, 30, 30),
998 1128 1 OPERATOR_ENTRY('/', DIVIDE, INFIX, 30, 30),
999 1129 1 OPERATOR_ENTRY('<', BITSELECT, POSTFIX, 50, 200, LEXICAL),
1000 1130 1 OPERATOR_ENTRY('(', OPENPAREN, PREFIX, 200, 5, LEXICAL),
1001 1131 1 OPERATOR_ENTRY(')', CLOSEPAREN, POSTFIX, 6, 200, LEXICAL));
1002 1132 1
1003 1133 1
1004 1134 1 ! Generate the Terminator Lexical Token Entry tables used for the EXAMINE
1005 1135 1 ! and other commands (EXAM A,B,C or EXAM A:B,C:D), the DEPOSIT command
1006 1136 1 ! (DEP X = Y), the SET BREAK command (SET BREAK X DO(...)), the IF command
1007 1137 1 ! (IF X THEN ...), and other situations (no terminator except end of command).
1008 1138 1
1009 1139 1 TERMINATOR_TABLE(EMPTY_TERM_TBL);
1010 1140 1
1011 1141 1 P TERMINATOR_TABLE(COMMA_TERM_TBL,
```

```
1012      1142 1      TERMINATOR_ENTRY(',', TERM_COMMA));
1013      1143 1
1014      P 1144 1      TERMINATOR_TABLE(EQUAL TERM_TBL,
1015      1145 1      TERMINATOR_ENTRY('=', TERM_EQUAL, BALANCED_PARENS));
1016      1146 1
1017      P 1147 1      TERMINATOR_TABLE(DO TERM_TBL,
1018      1148 1      TERMINATOR_ENTRY('DO', TERM_DO, BALANCED_PARENS));
1019      1149 1
1020      P 1150 1      TERMINATOR_TABLE(THEN TERM_TBL,
1021      1151 1      TERMINATOR_ENTRY('THEN', TERM_THEN, BALANCED_PARENS));
1022      1152 1
1023      P 1153 1      TERMINATOR_TABLE(COMCOL TERM_TBL,
1024      1154 1      TERMINATOR_ENTRY(':', TERM_COLON, MUST_BE_SINGLE));
1025      1155 1
1026      1156 1
1027      P 1157 1      TERMINATOR_TABLE(CMWHDO TERM_TBL,
1028      1158 1      TERMINATOR_ENTRY('CMWHDO', TERM_CMWHDO, BALANCED_PARENS),
1029      P 1159 1      TERMINATOR_ENTRY('WHEN', TERM_WHEN, BALANCED_PARENS),
1030      1160 1      TERMINATOR_ENTRY('DO', TERM_DO, BALANCED_PARENS));
1031      1161 1
1032      P 1162 1      TERMINATOR_TABLE(OPEN TERM_TBL,
1033      1163 1      TERMINATOR_ENTRY('T', TERM_OPEN));
1034      1164 1
1035      P 1165 1      TERMINATOR_TABLE(COMPAREN TERM_TBL,
1036      1166 1      TERMINATOR_ENTRY('P', TERM_COMMA),
1037      1167 1      TERMINATOR_ENTRY('}', TERM_CLOSE, BALANCED_PARENS));
1038      1168 1
1039      P 1169 1      TERMINATOR_TABLE(TO TERM_TBL,
1040      1170 1      TERMINATOR_ENTRY('TO', TERM_TO, BALANCED_PARENS));
1041      1171 1
1042      P 1172 1      TERMINATOR_TABLE(BY TERM_TBL,
1043      1173 1      TERMINATOR_ENTRY('BY', TERM_BY, BALANCED_PARENS),
1044      1174 1      TERMINATOR_ENTRY('DO', TERM_DO, BALANCED_PARENS));
1045      1175 1
1046      P 1176 1      TERMINATOR_TABLE(BIT SELECT TERM_TBL,
1047      1177 1      TERMINATOR_ENTRY('T', TERM_COMMA),
1048      1178 1      TERMINATOR_ENTRY('>', TERM_GTRTHAN));
1049      1179 1
1050      P 1180 1      TERMINATOR_TABLE(SET CONSTANT TERM_TBL,
1051      1181 1      TERMINATOR_ENTRY('C', TERM_COMMA),
1052      P 1182 1      TERMINATOR_ENTRY('}', TERM_CLOSE, BALANCED_PARENS),
1053      1183 1      TERMINATOR_ENTRY('.', TERM_DOT));
1054      1184 1
1055      1185 1
1056      1186 1      ! Generate a table indexed by Terminator Code which has pointers to the above
1057      1187 1      ! Terminator Lexical Token Entry tables. This table is used in DBG$EXP_INT
1058      1188 1      ! and DBG$ADDR_EXP_INT to look up the terminator table to be used for the
1059      1189 1      ! current expression of address expression.
1060      1190 1
1061      1191 1      OWN
1062      1192 1      TERM_POINTER_TBL: VECTOR[TOKEN$K_MAX_TERMINATOR + 1]
1063      1193 1      PSECT(DBG$PLIT) PRESET(
1064      1194 1      [TOKEN$K_TERM_NONE]      = EMPTY_TERM_TBL      - TABLEBASE,
1065      1195 1      [TOKEN$K_TERM_COMMA]      = COMMA_TERM_TBL      - TABLEBASE,
1066      1196 1      [TOKEN$K_TERM_EQUAL]      = EQUAL_TERM_TBL      - TABLEBASE,
1067      1197 1      [TOKEN$K_TERM_DO]        = DO_TERM_TBL        - TABLEBASE,
1068      1198 1      [TOKEN$K_TERM_THEN]      = THEN_TERM_TBL      - TABLEBASE,
```



```
1069 1199 1 [TOKEN$K_TERM_COMCOL] = COMCOL_TERM_TBL - TABLEBASE,  
1070 1200 1 [TOKEN$K_TERM_CMWHD0] = CMWHD0_TERM_TBL - TABLEBASE,  
1071 1201 1 [TOKEN$K_TERM_OPEN] = OPEN_TERM_TBL - TABLEBASE,  
1072 1202 1 [TOKEN$K_TERM_COMPAREN] = COMPAREN_TERM_TBL - TABLEBASE,  
1073 1203 1 [TOKEN$K_TERM_TO] = TO_TERM_TBL - TABLEBASE,  
1074 1204 1 [TOKEN$K_TERM_BY] = BY_TERM_TBL - TABLEBASE);  
1075 1205 1  
1076 1206 1  
1077 1207 1  
1078 1208 1  
1079 1209 1  
1080 1210 1  
1081 1211 1  
1082 1212 1  
1083 1213 1  
1084 1214 1  
1085 1215 1  
1086 1216 1  
1087 1217 1  
1088 1218 1  
1089 1219 1  
1090 1220 1  
1091 1221 1  
1092 1222 1  
1093 1223 1  
1094 1224 1  
1095 1225 1  
1096 1226 1  
1097 1227 1  
1098 1228 1  
1099 1229 1  
1100 1230 1  
1101 1231 1  
1102 1232 1  
1103 1233 1  
1104 1234 1  
1105 1235 1  
1106 1236 1  
1107 1237 1  
1108 1238 1  
1109 1239 1  
1110 1240 1  
1111 1241 1  
1112 1242 1  
1113 1243 1  
1114 1244 1  
1115 1245 1  
1116 1246 1  
1117 1247 1  
1118 1248 1  
1119 1249 1  
1120 1250 1  
1121 1251 1  
1122 1252 1  
1123 1253 1  
1124 1254 1  
1125 1255 1
```

Generate the base Character Table. The Character Table for each supported language is generated by using this Character Table as a base and then specifying a list of exceptions for specific characters. The result is a language-specific Character Table, generated in the CHARTBL vector.

```
CHAR_TABLE(BASE CHARACTER TABLE,  
  CHAR_ENT('A', HEXDIGIT, ALPHABETIC, IDENT_ANYWHERE),  
  CHAR_ENT('B', B, ALPHABETIC, IDENT_ANYWHERE, TERMINATOR),  
  CHAR_ENT('C', HEXDIGIT, ALPHABETIC, IDENT_ANYWHERE),  
  CHAR_ENT('D', D, ALPHABETIC, IDENT_ANYWHERE, TERMINATOR),  
  CHAR_ENT('E', E, ALPHABETIC, IDENT_ANYWHERE),  
  CHAR_ENT('F', HEXDIGIT, ALPHABETIC, IDENT_ANYWHERE),  
  CHAR_ENT('G', G, ALPHABETIC, IDENT_ANYWHERE),  
  CHAR_ENT('H', OTHER, ALPHABETIC, IDENT_ANYWHERE),  
  CHAR_ENT('I', OTHER, ALPHABETIC, IDENT_ANYWHERE),  
  CHAR_ENT('J', OTHER, ALPHABETIC, IDENT_ANYWHERE),  
  CHAR_ENT('K', OTHER, ALPHABETIC, IDENT_ANYWHERE),  
  CHAR_ENT('L', OTHER, ALPHABETIC, IDENT_ANYWHERE),  
  CHAR_ENT('M', OTHER, ALPHABETIC, IDENT_ANYWHERE),  
  CHAR_ENT('N', OTHER, ALPHABETIC, IDENT_ANYWHERE),  
  CHAR_ENT('O', OTHER, ALPHABETIC, IDENT_ANYWHERE),  
  CHAR_ENT('P', OTHER, ALPHABETIC, IDENT_ANYWHERE),  
  CHAR_ENT('Q', Q, ALPHABETIC, IDENT_ANYWHERE),  
  CHAR_ENT('R', OTHER, ALPHABETIC, IDENT_ANYWHERE),  
  CHAR_ENT('S', OTHER, ALPHABETIC, IDENT_ANYWHERE),  
  CHAR_ENT('T', OTHER, ALPHABETIC, IDENT_ANYWHERE, TERMINATOR),  
  CHAR_ENT('U', OTHER, ALPHABETIC, IDENT_ANYWHERE),  
  CHAR_ENT('V', OTHER, ALPHABETIC, IDENT_ANYWHERE),  
  CHAR_ENT('W', OTHER, ALPHABETIC, IDENT_ANYWHERE, TERMINATOR),  
  CHAR_ENT('X', X, ALPHABETIC, IDENT_ANYWHERE),  
  CHAR_ENT('Y', OTHER, ALPHABETIC, IDENT_ANYWHERE),  
  CHAR_ENT('Z', OTHER, ALPHABETIC, IDENT_ANYWHERE),  
  
  CHAR_ENT('a', HEXDIGIT, ALPHABETIC, IDENT_ANYWHERE),  
  CHAR_ENT('b', B, ALPHABETIC, IDENT_ANYWHERE),  
  CHAR_ENT('c', HEXDIGIT, ALPHABETIC, IDENT_ANYWHERE),  
  CHAR_ENT('d', D, ALPHABETIC, IDENT_ANYWHERE, TERMINATOR),  
  CHAR_ENT('e', E, ALPHABETIC, IDENT_ANYWHERE),  
  CHAR_ENT('f', HEXDIGIT, ALPHABETIC, IDENT_ANYWHERE),  
  CHAR_ENT('g', G, ALPHABETIC, IDENT_ANYWHERE),  
  CHAR_ENT('h', OTHER, ALPHABETIC, IDENT_ANYWHERE),  
  CHAR_ENT('i', OTHER, ALPHABETIC, IDENT_ANYWHERE),  
  CHAR_ENT('j', OTHER, ALPHABETIC, IDENT_ANYWHERE),  
  CHAR_ENT('k', OTHER, ALPHABETIC, IDENT_ANYWHERE),  
  CHAR_ENT('l', OTHER, ALPHABETIC, IDENT_ANYWHERE),  
  CHAR_ENT('m', OTHER, ALPHABETIC, IDENT_ANYWHERE),  
  CHAR_ENT('n', OTHER, ALPHABETIC, IDENT_ANYWHERE),  
  CHAR_ENT('o', OTHER, ALPHABETIC, IDENT_ANYWHERE),  
  CHAR_ENT('p', OTHER, ALPHABETIC, IDENT_ANYWHERE),
```

1126	P	1256	1	CHAR-ENT('q',	Q,	ALPHABETIC, IDENT_ANYWHERE),
1127	P	1257	1	CHAR-ENT('r',	OTHER,	ALPHABETIC, IDENT_ANYWHERE),
1128	P	1258	1	CHAR-ENT('s',	OTHER,	ALPHABETIC, IDENT_ANYWHERE),
1129	P	1259	1	CHAR-ENT('t',	OTHER,	ALPHABETIC, IDENT_ANYWHERE, TERMINATOR),
1130	P	1260	1	CHAR-ENT('u',	OTHER,	ALPHABETIC, IDENT_ANYWHERE),
1131	P	1261	1	CHAR-ENT('v',	OTHER,	ALPHABETIC, IDENT_ANYWHERE),
1132	P	1262	1	CHAR-ENT('w',	OTHER,	ALPHABETIC, IDENT_ANYWHERE, TERMINATOR),
1133	P	1263	1	CHAR-ENT('x',	X,	ALPHABETIC, IDENT_ANYWHERE),
1134	P	1264	1	CHAR-ENT('y',	OTHER,	ALPHABETIC, IDENT_ANYWHERE),
1135	P	1265	1	CHAR-ENT('z',	OTHER,	ALPHABETIC, IDENT_ANYWHERE),
1136	P	1266	1			
1137	P	1267	1	CHAR-ENT('0',	DIGIT,	DIGIT, IDENT_MIDDLE, IDENT_END, NUMBER_START),
1138	P	1268	1	CHAR-ENT('1',	DIGIT,	DIGIT, IDENT_MIDDLE, IDENT_END, NUMBER_START),
1139	P	1269	1	CHAR-ENT('2',	DIGIT,	DIGIT, IDENT_MIDDLE, IDENT_END, NUMBER_START),
1140	P	1270	1	CHAR-ENT('3',	DIGIT,	DIGIT, IDENT_MIDDLE, IDENT_END, NUMBER_START),
1141	P	1271	1	CHAR-ENT('4',	DIGIT,	DIGIT, IDENT_MIDDLE, IDENT_END, NUMBER_START),
1142	P	1272	1	CHAR-ENT('5',	DIGIT,	DIGIT, IDENT_MIDDLE, IDENT_END, NUMBER_START),
1143	P	1273	1	CHAR-ENT('6',	DIGIT,	DIGIT, IDENT_MIDDLE, IDENT_END, NUMBER_START),
1144	P	1274	1	CHAR-ENT('7',	DIGIT,	DIGIT, IDENT_MIDDLE, IDENT_END, NUMBER_START),
1145	P	1275	1	CHAR-ENT('8',	DIGIT,	DIGIT, IDENT_MIDDLE, IDENT_END, NUMBER_START),
1146	P	1276	1	CHAR-ENT('9',	DIGIT,	DIGIT, IDENT_MIDDLE, IDENT_END, NUMBER_START),
1147	P	1277	1			
1148	P	1278	1	CHAR-ENT(' ',	OTHER,	SPACE),
1149	P	1279	1	CHAR-ENT('9',	OTHER,	SPACE),
1150	P	1280	1	CHAR-ENT('!',	OTHER,	! Tab character
1151	P	1281	1	CHAR-ENT('!',	OTHER,	NOTHING),
1152	P	1282	1	CHAR-ENT('!',	OTHER,	STRING QUOTE),
1153	P	1283	1	CHAR-ENT('#',	OTHER,	NOTHING),
1154	P	1284	1	CHAR-ENT('\$',	OTHER,	IDENT_MIDDLE, IDENT_END),
1155	P	1285	1	CHAR-ENT('%',	OTHER,	NOTHING),
1156	P	1286	1	CHAR-ENT('&',	OTHER,	NOTHING),
1157	P	1287	1	CHAR-ENT('(',	OTHER,	STRING QUOTE),
1158	P	1288	1	CHAR-ENT('(',	OTHER,	OPCHAR, ADDRESS_OP, TERMINATOR),
1159	P	1289	1	CHAR-ENT('(',	OTHER,	OPCHAR, ADDRESS_OP, TERMINATOR),
1160	P	1290	1	CHAR-ENT('*',	OTHER,	OPCHAR, OPCHAR INFIX, ADDRESS_OP),
1161	P	1291	1	CHAR-ENT('+',	PLUS,	OPCHAR, ADDRESS_OP),
1162	P	1292	1	CHAR-ENT('-',	OTHER,	TERMINATOR),
1163	P	1293	1	CHAR-ENT('-',	MINUS,	OPCHAR, ADDRESS_OP),
1164	P	1294	1	CHAR-ENT('.',	DOT,	NUMBER_START, OPCHAR, ADDRESS_OP, SPECIAL_SYMBOL),
1165	P	1295	1	CHAR-ENT('/',	OTHER,	OPCHAR, ADDRESS_OP),
1166	P	1296	1	CHAR-ENT(':',	OTHER,	TERMINATOR),
1167	P	1297	1	CHAR-ENT(':',	OTHER,	NOTHING),
1168	P	1298	1	CHAR-ENT('<',	OTHER,	ADDRESS_OP),
1169	P	1299	1	CHAR-ENT('=',	OTHER,	TERMINATOR),
1170	P	1300	1	CHAR-ENT('>',	OTHER,	TERMINATOR),
1171	P	1301	1	CHAR-ENT('?',	OTHER,	NOTHING),
1172	P	1302	1	CHAR-ENT('@',	OTHER,	OPCHAR, ADDRESS_OP),
1173	P	1303	1	CHAR-ENT('[',	OTHER,	NOTHING),
1174	P	1304	1	CHAR-ENT('\',	OTHER,	OPCHAR, SPECIAL_SYMBOL),
1175	P	1305	1	CHAR-ENT(']',	OTHER,	NOTHING),
1176	P	1306	1	CHAR-ENT('^',	OTHER,	SPECIAL_SYMBOL),
1177	P	1307	1	CHAR-ENT('_',	OTHER,	IDENT_MIDDLE, IDENT_END),
1178	P	1308	1	CHAR-ENT('`',	OTHER,	NOTHING),
1179	P	1309	1	CHAR-ENT('`',	OTHER,	NOTHING),
1180	P	1310	1	CHAR-ENT('`',	OTHER,	NOTHING),
1181		1311	1	CHAR-ENT('`',	OTHER,	NOTHING);

LANGUAGE UNKNOWN PARSE TABLES

This section contains all Lexical Scanner and Parser tables for language UNKNOWN. Language "UNKNOWN" constitutes the language facilities made available by DEBUG when the actual source language is not known to DEBUG. The main use for language UNKNOWN is to provide a basic level of DEBUG support for new languages which are not yet explicitly supported by DEBUG with its own syntax and semantics.

(Language UNKNOWN comes first because some of the other languages make references to the UNKNOWN number table. The rest of the languages are in alphabetical order.)

Define the language UNKNOWN Character Table.

```
CHAR_EXCEPTION TABLE(UNKNOWN CHARTBL,  
  CHAR_ENTRY('8', OTHER, OPCHAR),  
  CHAR_ENTRY('/', OTHER, OPCHAR, OPCHAR_INFIX, ADDRESS_OP),  
  CHAR_ENTRY('<', OTHER, OPCHAR, OPCHAR_INFIX, ADDRESS_OP),  
  CHAR_ENTRY('=', OTHER, OPCHAR, OPCHAR_INFIX, TERMINATOR),  
  CHAR_ENTRY('>', OTHER, OPCHAR, OPCHAR_INFIX, TERMINATOR),  
  CHAR_ENTRY('^', OTHER, OPCHAR, SPECIAL_SYMBOL),  
  CHAR_ENTRY('[', OTHER, OPCHAR),  
  CHAR_ENTRY(']', OTHER, TERMINATOR));
```

Define the language UNKNOWN Operator Table for operators whose names are identifiers.

```
OPERATOR TABLE(UNKNOWN IDENT_OPTBL,  
  OPERATOR_ENTRY('EQL', EQUAL, INFIX, 50, 50),  
  OPERATOR_ENTRY('NEQ', NOT_EQUAL, INFIX, 50, 50),  
  OPERATOR_ENTRY('GTR', GTR_THAN, INFIX, 50, 50),  
  OPERATOR_ENTRY('GEQ', GTR_EQUAL, INFIX, 50, 50),  
  OPERATOR_ENTRY('LSS', LSS_THAN, INFIX, 50, 50),  
  OPERATOR_ENTRY('LEQ', LSS_EQUAL, INFIX, 50, 50),  
  OPERATOR_ENTRY('NOT', NOT, PREFIX, 200, 40),  
  OPERATOR_ENTRY('AND', AND, INFIX, 30, 30),  
  OPERATOR_ENTRY('OR', OR, INFIX, 20, 20),  
  OPERATOR_ENTRY('XOR', XOR, INFIX, 10, 10),  
  OPERATOR_ENTRY('EQV', EQV, INFIX, 10, 10));
```

Define the language UNKNOWN Operator Table for operators whose names are composed of operator characters such as '+' or '*'. This table includes operators which are part of Primary Symbols (such as '\' and '.').

```
OPERATOR TABLE(UNKNOWN OPCHAR_OPTBL,  
  OPERATOR_ENTRY('\', GLOBAL_SLASH, PREFIX, 0, 0, PRIMARY),  
  OPERATOR_ENTRY('.', BACKSLASH, INFIX, 0, 0, PRIMARY),  
  OPERATOR_ENTRY('.', DOT, INFIX, 0, 0, PRIMARY),  
  OPERATOR_ENTRY('(', SUBSCRIPT, POSTFIX, 0, 0, PRIMARY),  
  OPERATOR_ENTRY('[', SUBSCRIPT, POSTFIX, 0, 0, PRIMARY),
```



```
1240 P 1369 1 OPERATOR_ENTRY('`', PASCAL_DEREF, POSTFIX, 0, 0, PRIMARY),
1241 P 1370 1
1242 P 1371 1 OPERATOR_ENTRY('//', CONCATENATE, INFIX, 60, 60),
1243 P 1372 1 OPERATOR_ENTRY('&', CONCATENATE, INFIX, 60, 60),
1244 P 1373 1 OPERATOR_ENTRY('+', ADD, INFIX, 60, 60),
1245 P 1374 1 OPERATOR_ENTRY('-', SUBTRACT, INFIX, 60, 60),
1246 P 1375 1 OPERATOR_ENTRY('++', UNARY_PLUS, PREFIX, 200, 70),
1247 P 1376 1 OPERATOR_ENTRY('--', UNARY_MINUS, PREFIX, 200, 70),
1248 P 1377 1 OPERATOR_ENTRY('*', MULTIPLY, INFIX, 80, 80),
1249 P 1378 1 OPERATOR_ENTRY('/', DIVIDE, INFIX, 80, 80),
1250 P 1379 1 OPERATOR_ENTRY('**', POWER_OF, INFIX, 92, 90),
1251 P 1380 1 OPERATOR_ENTRY('=', EQUAL, INFIX, 50, 50),
1252 P 1381 1 OPERATOR_ENTRY('<>', NOT_EQUAL, INFIX, 50, 50),
1253 P 1382 1 OPERATOR_ENTRY('/=', NOT_EQUAL, INFIX, 50, 50),
1254 P 1383 1 OPERATOR_ENTRY('>', GTR_THAN, INFIX, 50, 50),
1255 P 1384 1 OPERATOR_ENTRY('>=', GTR_EQUAL, INFIX, 50, 50),
1256 P 1385 1 OPERATOR_ENTRY('<', LSS_THAN, INFIX, 50, 50),
1257 P 1386 1 OPERATOR_ENTRY('<=', LSS_EQUAL, INFIX, 50, 50),
1258 P 1387 1 OPERATOR_ENTRY('(', OPENPAREN, PREFIX, 200, 5, LEXICAL),
1259 P 1388 1 OPERATOR_ENTRY(')', CLOSEPAREN, POSTFIX, 6, 200, LEXICAL));
1260 P 1389 1
1261 P 1390 1
1262 P 1391 1 ! Define the UNKNOWN Terminator Lexical Token Table for subscript expressions.
1263 P 1392 1 ! Here we allow subscript expressions to be terminated by ')' (end of sub-
1264 P 1393 1 ! scripts) and by ',' (more subscripts to follow).
1265 P 1394 1
1266 P 1395 1 TERMINATOR_TABLE(UNKNOWN SUBSCR TERM TBL,
1267 P 1396 1 TERMINATOR_ENTRY(')', TERM_CLOSE, BALANCED_PARENS),
1268 P 1397 1 TERMINATOR_ENTRY(',', TERM_CLOSE, BALANCED_PARENS),
1269 P 1398 1 TERMINATOR_ENTRY(':', TERM_COLON, MUST_BE_SINGLE),
1270 P 1399 1 TERMINATOR_ENTRY('.', TERM_COMMA));
1271 P 1400 1
1272 P 1401 1
1273 P 1402 1 ! Define the language UNKNOWN Number Scanner State Table. This is a finite-state
1274 P 1403 1 ! machine in which each transition is of the form:
1275 P 1404 1
1276 P 1405 1 NUMBER_TRANSITION(character-class, action-index, next-state)
1277 P 1406 1
1278 P 1407 1 ! where the character-class and action-index names are automatically prefixed
1279 P 1408 1 ! by 'NUMSTSK_CLASS_' or 'NUMSTSK_ACT_' by the NUMBER_TRANSITION macro.
1280 P 1409 1
1281 P 1410 1 NUMBER_STATE_TABLE(UNKNOWN_NUMBER_TABLE,
1282 P 1411 1
1283 P 1412 1 NUMBER_STATE(START_STATE,
1284 P 1413 1 NUMBER_TRANSITION(DIGIT, GO_PAST_DIGIT, ACCUM_INT),
1285 P 1414 1 NUMBER_TRANSITION(DOT, MARK_DEC_PT, LEADING_DOT),
1286 P 1415 1 NUMBER_TRANSITION(OTHER, NOT_NUMBER, END_STATE)),
1287 P 1416 1
1288 P 1417 1 NUMBER_STATE(LEADING_DOT,
1289 P 1418 1 NUMBER_TRANSITION(DIGIT, GO_PAST_FRAC, ACCUM_FRAC),
1290 P 1419 1 NUMBER_TRANSITION(OTHER, NOT_NUMBER, END_STATE)),
1291 P 1420 1
1292 P 1421 1 NUMBER_STATE(ACCUM_INT,
1293 P 1422 1 NUMBER_TRANSITION(DIGIT, GO_PAST_DIGIT, ACCUM_INT),
1294 P 1423 1 NUMBER_TRANSITION(DOT, MARK_DEC_PT, ACCUM_FRAC),
1295 P 1424 1 NUMBER_TRANSITION(HEXDIGIT, DO_NOTHING, ACCUM_HEX),
1296 P 1425 1 NUMBER_TRANSITION(B, DO_NOTHING, ACCUM_HEX),
```

```
1297 P 1426 1 NUMBER_TRANSITION(D, DO_NOTHING, ACCUM_HEX),
1298 P 1427 1 NUMBER_TRANSITION(E, DO_NOTHING, ACCUM_HEX),
1299 P 1428 1 NUMBER_TRANSITION(OTHER, GOT_NUMBER, END_STATE)),
1300 P 1429 1
1301 P 1430 1 NUMBER_STATE(ACCUM_HEX,
1302 P 1431 1 NUMBER_TRANSITION(DIGIT, DO_NOTHING, ACCUM_HEX),
1303 P 1432 1 NUMBER_TRANSITION(HEXDIGIT, DO_NOTHING, ACCUM_HEX),
1304 P 1433 1 NUMBER_TRANSITION(B, DO_NOTHING, ACCUM_HEX),
1305 P 1434 1 NUMBER_TRANSITION(D, DO_NOTHING, ACCUM_HEX),
1306 P 1435 1 NUMBER_TRANSITION(E, DO_NOTHING, ACCUM_HEX),
1307 P 1436 1 NUMBER_TRANSITION(OTHER, GOT_NUMBER, END_STATE)),
1308 P 1437 1
1309 P 1438 1 NUMBER_STATE(ACCUM_FRAC,
1310 P 1439 1 NUMBER_TRANSITION(DIGIT, GO_PAST_FRAC, ACCUM_FRAC),
1311 P 1440 1 NUMBER_TRANSITION(DOT, BACKUP_PTRS, END_STATE),
1312 P 1441 1 NUMBER_TRANSITION(E, MARK_E_EXP, GET_EXPONENT),
1313 P 1442 1 NUMBER_TRANSITION(D, MARK_D_EXP, GET_EXPONENT),
1314 P 1443 1 NUMBER_TRANSITION(G, MARK_G_EXP, GET_EXPONENT),
1315 P 1444 1 NUMBER_TRANSITION(Q, MARK_Q_EXP, GET_EXPONENT),
1316 P 1445 1 NUMBER_TRANSITION(OTHER, GOT_NUMBER, END_STATE)),
1317 P 1446 1
1318 P 1447 1 NUMBER_STATE(GET_EXPONENT,
1319 P 1448 1 NUMBER_TRANSITION(DIGIT, DO_NOTHING, ACCUM_EXP),
1320 P 1449 1 NUMBER_TRANSITION(PLUS, DO_NOTHING, GET_EXP_SIGN),
1321 P 1450 1 NUMBER_TRANSITION(MINUS, DO_NOTHING, GET_EXP_SIGN),
1322 P 1451 1 NUMBER_TRANSITION(OTHER, BACKUP_PTRS, END_STATE)),
1323 P 1452 1
1324 P 1453 1 NUMBER_STATE(GET_EXP_SIGN,
1325 P 1454 1 NUMBER_TRANSITION(DIGIT, DO_NOTHING, ACCUM_EXP),
1326 P 1455 1 NUMBER_TRANSITION(OTHER, GOT_NUMBER, END_STATE)),
1327 P 1456 1
1328 P 1457 1 NUMBER_STATE(ACCUM_EXP,
1329 P 1458 1 NUMBER_TRANSITION(DIGIT, DO_NOTHING, ACCUM_EXP),
1330 P 1459 1 NUMBER_TRANSITION(OTHER, GOT_NUMBER, END_STATE)),
1331 P 1460 1
1332 P 1461 1 NUMBER_STATE(END_STATE,
1333 1462 1 NUMBER_TRANSITION(OTHER, GIVE_ERROR, END_STATE));
1334 1463 1
1335 1464 1
1336 1465 1 ! Define the language UNKNOWN Predefined Identifier Table.
1337 1466 1
1338 1467 1 PRID_TABLE(UNKNOWN_PRID_TABLE);
1339 1468 1
1340 1469 1
1341 1470 1 ! Define the language UNKNOWN Built-in Function Table.
1342 1471 1
1343 1472 1 BUILT_IN_FUNCTION_TABLE(UNKNOWN_FUNCTION_TABLE);
1344 1473 1
1345 1474 1
1346 1475 1 ! Define the Primary Parser State Table for language UNKNOWN.
1347 1476 1
1348 P 1477 1 PRIMARY_STATE_TABLE(UNKNOWN_PRIMARY_TABLE,
1349 P 1478 1
1350 P 1479 1 PRIMARY_STATE(START_STATE,
1351 P 1480 1 PRIMARY_TRANSITION(GLOBAL_SLASH, START_GBL, GET_GLOBAL),
1352 P 1481 1 PRIMARY_TRANSITION(BACKSLASH, START_SLASH, GOT_BACKSLASH),
1353 P 1482 1 PRIMARY_TRANSITION(INVOCNUM, SLASH_INVOCNUM, GOT_BACKSLASH),
```

```
1354 P 1483 1 PRIMARY_TRANSITION(DOT, START_DOT, GOT_DOT),
1355 P 1484 1 PRIMARY_TRANSITION(SUBSCRIPT, START_SUBSCR, GOT_SUBSCRIPT),
1356 P 1485 1 PRIMARY_TRANSITION(PASCAL_DEREF, START_DEREF, GOT_DEREF),
1357 P 1486 1 PRIMARY_TRANSITION(PRIMARY_TERM, START_TERM, END_STATE)),
1358 P 1487 1
1359 P 1488 1 PRIMARY_STATE(GET_GLOBAL,
1360 P 1489 1 PRIMARY_TRANSITION(PRIMARY_TERM, GBL_TERM, END_STATE)),
1361 P 1490 1
1362 P 1491 1 PRIMARY_STATE(GOT_BACKSLASH,
1363 P 1492 1 PRIMARY_TRANSITION(BACKSLASH, SLASH_SLASH, GOT_BACKSLASH),
1364 P 1493 1 PRIMARY_TRANSITION(INVOCNUM, SLASH_INVOCNUM, GOT_BACKSLASH),
1365 P 1494 1 PRIMARY_TRANSITION(DOT, SLASH_DOT, GOT_DOT),
1366 P 1495 1 PRIMARY_TRANSITION(SUBSCRIPT, SLASH_SUBSCR, GOT_SUBSCRIPT),
1367 P 1496 1 PRIMARY_TRANSITION(PASCAL_DEREF, SLASH_DEREF, GOT_DEREF),
1368 P 1497 1 PRIMARY_TRANSITION(PRIMARY_TERM, SLASH_TERM, END_STATE)),
1369 P 1498 1
1370 P 1499 1 PRIMARY_STATE(GOT_DOT,
1371 P 1500 1 PRIMARY_TRANSITION(DOT, DOT_DOT, GOT_DOT),
1372 P 1501 1 PRIMARY_TRANSITION(SUBSCRIPT, DOT_SUBSCR, GOT_SUBSCRIPT),
1373 P 1502 1 PRIMARY_TRANSITION(PASCAL_DEREF, DOT_DEREF, GOT_DEREF),
1374 P 1503 1 PRIMARY_TRANSITION(PRIMARY_TERM, DOT_TERM, END_STATE)),
1375 P 1504 1
1376 P 1505 1 PRIMARY_STATE(GOT_SUBSCRIPT,
1377 P 1506 1 PRIMARY_TRANSITION(DOT, SUBSCR_DOT, GOT_DOT),
1378 P 1507 1 PRIMARY_TRANSITION(SUBSCRIPT, SUBSCR_SUBSCR, GOT_SUBSCRIPT),
1379 P 1508 1 PRIMARY_TRANSITION(PASCAL_DEREF, SUBSCR_DEREF, GOT_DEREF),
1380 P 1509 1 PRIMARY_TRANSITION(PRIMARY_TERM, SUBSCR_TERM, END_STATE)),
1381 P 1510 1
1382 P 1511 1 PRIMARY_STATE(GOT_DEREF,
1383 P 1512 1 PRIMARY_TRANSITION(DOT, DEREF_DOT, GOT_DOT),
1384 P 1513 1 PRIMARY_TRANSITION(SUBSCRIPT, DEREF_SUBSCR, GOT_SUBSCRIPT),
1385 P 1514 1 PRIMARY_TRANSITION(PASCAL_DEREF, DEREF_DEREF, GOT_DEREF),
1386 P 1515 1 PRIMARY_TRANSITION(PRIMARY_TERM, DEREF_TERM, END_STATE)),
1387 P 1516 1
1388 1517 1 PRIMARY_STATE(END_STATE));
1389 1518 1
1390 1519 1
1391 1520 1 ! Finally define the table of pointers to the parse tables for language UNKNOWN.
1392 1521 1 !
1393 P 1522 1 LANGUAGE_TABLES(LANGUAGE = UNKNOWN,
1394 P 1523 1 CHARTBL = UNKNOWN CHARTBL,
1395 P 1524 1 IDENT_OPTBL = UNKNOWN IDENT_OPTBL,
1396 P 1525 1 OPCHAR_OPTBL = UNKNOWN OPCHAR_OPTBL,
1397 P 1526 1 NUMBER_TABLE = UNKNOWN NUMBER_TABLE,
1398 P 1527 1 PRIMARY_TABLE = UNKNOWN PRIMARY_TABLE,
1399 P 1528 1 SUBSCR_TERMS = UNKNOWN SUBSCR_TERM_TBL,
1400 P 1529 1 PRIDTBL = UNKNOWN PRID_TABLE,
1401 1530 1 BIF_TABLE = UNKNOWN_FUNCTION_TABLE);
```


A D A P A R S E T A B L E S

This section includes all the Lexical Scanner and Parser tables needed to scan and parse the ADA language.

Define the ADA Character Table. What is listed here is actually a list of exceptions to the Character Table for Language UNKNOWN.

```
CHAR_EXCEPTION_TABLE(ADA_CHARTBL,  
  CHAR_ENTRY(' ', OTHER, NOTHING),  
  CHAR_ENTRY(' ', UNDERSCORE, IDENT_MIDDLE),  
  CHAR_ENTRY(' ', POUND, NOTHING),  
  CHAR_ENTRY(' ', OTHER, OPCHAR),  
  CHAR_ENTRY('/', OTHER, OPCHAR, OPCHAR_INFIX, ADDRESS_OP),  
  CHAR_ENTRY('<', OTHER, OPCHAR, OPCHAR_INFIX, ADDRESS_OP),  
  CHAR_ENTRY('=', OTHER, OPCHAR, OPCHAR_INFIX, TERMINATOR),  
  CHAR_ENTRY('>', OTHER, OPCHAR, OPCHAR_INFIX, TERMINATOR),  
  CHAR_ENTRY('.', DOT, OPCHAR, ADDRESS_OP, SPECIAL_SYMBOL, TERMINATOR));
```

Define the ADA Operator Table for operators whose names are identifiers.

```
OPERATOR_TABLE(ADA_IDENT_OPTBL,  
  OPERATOR_ENTRY('NOT', NOT, PREFIX, 200, 60),  
  OPERATOR_ENTRY('ABS', ABSOLUTE, PREFIX, 200, 60),  
  OPERATOR_ENTRY('MOD', MODULUS, INFIX, 50, 50),  
  OPERATOR_ENTRY('REM', REMAINDER, INFIX, 50, 50),  
  OPERATOR_ENTRY('AND', AND, INFIX, 10, 10),  
  OPERATOR_ENTRY('OR', OR, INFIX, 10, 10),  
  OPERATOR_ENTRY('XOR', XOR, INFIX, 10, 10));
```

Define the ADA Operator Table for operators whose names are composed of operator characters such as '+', '-', or '*'. This table includes operators which are part of DEBUG Primary Symbols (such as '\').

```
OPERATOR_TABLE(ADA_OPCHAR_OPTBL,  
  OPERATOR_ENTRY('\ ', GLOBAL_SLASH, PREFIX, 0, 0, PRIMARY),  
  OPERATOR_ENTRY('\ ', BACKSLASH, INFIX, 0, 0, PRIMARY),  
  OPERATOR_ENTRY('.', DOT, INFIX, 0, 0, PRIMARY),  
  OPERATOR_ENTRY(' ', SUBSCRIPT, POSTFIX, 0, 0, PRIMARY),  
  
  OPERATOR_ENTRY('(', OPENPAREN, PREFIX, 200, 5, LEXICAL),  
  OPERATOR_ENTRY(')', CLOSEPAREN, POSTFIX, 6, 200, LEXICAL),  
  OPERATOR_ENTRY('**', POWER_OF, INFIX, 60, 60),  
  OPERATOR_ENTRY('*', MULTIPLY, INFIX, 50, 50),  
  OPERATOR_ENTRY('/', DIVIDE, INFIX, 50, 50),  
  OPERATOR_ENTRY('+', UNARY_PLUS, PREFIX, 200, 40),  
  OPERATOR_ENTRY('-', UNARY_MINUS, PREFIX, 200, 40),  
  OPERATOR_ENTRY('+', ADD, INFIX, 30, 30),  
  OPERATOR_ENTRY('-', SUBTRACT, INFIX, 30, 30),  
  OPERATOR_ENTRY('&', CONCATENATE, INFIX, 30, 30),  
  OPERATOR_ENTRY('=', EQUAL, INFIX, 20, 20).
```

```
1460 P 1588 1 OPERATOR_ENTRY('/=', NOT_EQUAL, INFIX, 20, 20),
1461 P 1589 1 OPERATOR_ENTRY('<', LSS_THAN, INFIX, 20, 20),
1462 P 1590 1 OPERATOR_ENTRY('<=', LSS_EQUAL, INFIX, 20, 20),
1463 P 1591 1 OPERATOR_ENTRY('>', GTR_THAN, INFIX, 20, 20),
1464 1592 1 OPERATOR_ENTRY('>=', GTR_EQUAL, INFIX, 20, 20));
1465 1593 1
1466 1594 1
1467 1595 1 ! Define an Operator Lexical Token Entry for the ADA Tick operator '""'.
1468 1596 1 ! This token is actually used to represent a collection of postfix
1469 1597 1 ! tick operators ('FIRST', 'LAST', ...). The lexical scanner will
1470 1598 1 ! fill in the subcode field which identifies which tick operator was
1471 1599 1 ! given.
1472 1600 1
1473 1601 1 BIND
1474 1602 1 ADA_TICK_TOKEN = OPERATOR_ENTRY('""', ADA_TICK, POSTFIX, 0, 0, PRIMARY);
1475 1603 1
1476 1604 1
1477 1605 1 OWN
1478 1606 1 ADA_TICK_TABLE: VECTOR [TOKENSK_TICK_MAX+1] PSECT(DBG$PLIT) PRESET (
1479 1607 1 [TOKENSK_TICK_CONSTRAINED] = UPLIT (%ASCIC 'CONSTRAINED') - TABLEBASE,
1480 1608 1 [TOKENSK_TICK_FIRST] = UPLIT (%ASCIC 'FIRST') - TABLEBASE,
1481 1609 1 [TOKENSK_TICK_LAST] = UPLIT (%ASCIC 'LAST') - TABLEBASE,
1482 1610 1 [TOKENSK_TICK_LENGTH] = UPLIT (%ASCIC 'LENGTH') - TABLEBASE,
1483 1611 1 [TOKENSK_TICK_POS] = UPLIT (%ASCIC 'POS') - TABLEBASE,
1484 1612 1 [TOKENSK_TICK_PRED] = UPLIT (%ASCIC 'PRED') - TABLEBASE,
1485 1613 1 [TOKENSK_TICK_SIZE] = UPLIT (%ASCIC 'SIZE') - TABLEBASE,
1486 1614 1 [TOKENSK_TICK_SUCC] = UPLIT (%ASCIC 'SUCC') - TABLEBASE,
1487 1615 1 [TOKENSK_TICK_VAL] = UPLIT (%ASCIC 'VAL') - TABLEBASE);
1488 1616 1
1489 1617 1 ! Define the ADA Terminator Lexical Token Table for subscript expressions.
1490 1618 1
1491 P 1619 1 TERMINATOR_TABLE(ADA_SUBSCR_TERM_TBL,
1492 P 1620 1 TERMINATOR_ENTRY(')', TERM_CLOSE),
1493 P 1621 1 TERMINATOR_ENTRY(':', TERM_COLON, MUST_BE_SINGLE),
1494 P 1622 1 TERMINATOR_ENTRY(';', TERM_COLON),
1495 1623 1 TERMINATOR_ENTRY(',', TERM_COMMA));
1496 1624 1
1497 1625 1
1498 1626 1 ! Define the ADA Predefined Identifier Table.
1499 1627 1
1500 1628 1 PRID_TABLE(ADA_PRID_TABLE);
1501 1629 1
1502 1630 1
1503 1631 1 ! Define the ADA Built-in Function Table.
1504 1632 1
1505 1633 1 BUILT_IN_FUNCTION_TABLE(ADA_FUNCTION_TABLE);
1506 1634 1
1507 1635 1
1508 1636 1 ! Define the ADA Number Scanner State Table. This table defines the states
1509 1637 1 ! of a Finite-State Machine which picks up all valid numeric constants in the
1510 1638 1 ! language.
1511 1639 1
1512 1640 1 ! Each transition is of the form:
1513 1641 1 ! NUMBER_TRANSITION(character-class, action-index, next-state)
1514 1642 1
1515 1643 1 ! The ADA standard defines a number to be of the form:
1516 1644 1 !
```

```
1517 1645 1 number_literal ::= integer [.integer] [exponent]
1518 1646 1 integer ::= digit { [underline] digit }
1519 1647 1 exponent ::= E [+] integer | E - integer
1520 1648 1
1521 1649 1 Examples:
1522 1650 1
1523 1651 1 12 1E6 123.456 integer literals
1524 1652 1 12.0 3.141_592 1.34E+6 real literals
1525 1653 1
1526 1654 1 The ADA standard also defines a based number to be of the form:
1527 1655 1
1528 1656 1 base # based_integer [.based_integer] # [exponent]
1529 1657 1 base ::= integer
1530 1658 1 based_integer ::= extended_digit { [underline] extended_digit }
1531 1659 1 extended_digit ::= digit | letter
1532 1660 1
1533 1661 1 Examples:
1534 1662 1
1535 1663 1 2#1111.1111# 16#FF# integer literals
1536 1664 1 16#FF.FF#E+2 real literal
1537 1665 1
1538 P 1666 1 NUMBER_STATE_TABLE(ADA_NUMBER_TABLE,
1539 P 1667 1
1540 P 1668 1 NUMBER_STATE(START_STATE,
1541 P 1669 1 NUMBER_TRANSITION(DIGIT, DO NOTHING, ACCUM_INT),
1542 P 1670 1 NUMBER_TRANSITION(OTHER, NOT_NUMBER, END_STATE)),
1543 P 1671 1
1544 P 1672 1 NUMBER_STATE(ACCUM_INT,
1545 P 1673 1 NUMBER_TRANSITION(DIGIT, DO NOTHING, ACCUM_INT),
1546 P 1674 1 NUMBER_TRANSITION(UNDERSCORE, DO NOTHING, T_ACCUM_INT),
1547 P 1675 1 NUMBER_TRANSITION(POUND, SAVE_BASE, B_START_STATE),
1548 P 1676 1 NUMBER_TRANSITION(DOT, MARK_DEC_PT, ACCUM_FRAC),
1549 P 1677 1 NUMBER_TRANSITION(E, DO NOTHING, GET_EXPONENT),
1550 P 1678 1 NUMBER_TRANSITION(OTHER, GOT_NUMBER, END_STATE)),
1551 P 1679 1
1552 P 1680 1 NUMBER_STATE(T_ACCUM_INT,
1553 P 1681 1 NUMBER_TRANSITION(DIGIT, DO NOTHING, ACCUM_INT),
1554 P 1682 1 NUMBER_TRANSITION(OTHER, NOT_NUMBER, END_STATE)),
1555 P 1683 1
1556 P 1684 1 NUMBER_STATE(ACCUM_FRAC,
1557 P 1685 1 NUMBER_TRANSITION(DIGIT, DO NOTHING, ACCUM_FRAC),
1558 P 1686 1 NUMBER_TRANSITION(UNDERSCORE, DO NOTHING, T_ACCUM_FRAC),
1559 P 1687 1 NUMBER_TRANSITION(DOT, NOT_NUMBER, END_STATE),
1560 P 1688 1 NUMBER_TRANSITION(E, MARK_E_EXP, GET_EXPONENT),
1561 P 1689 1 NUMBER_TRANSITION(D, MARK_D_EXP, GET_EXPONENT),
1562 P 1690 1 NUMBER_TRANSITION(G, MARK_G_EXP, GET_EXPONENT),
1563 P 1691 1 NUMBER_TRANSITION(Q, MARK_Q_EXP, GET_EXPONENT),
1564 P 1692 1 NUMBER_TRANSITION(OTHER, GOT_NUMBER, END_STATE)),
1565 P 1693 1
1566 P 1694 1 NUMBER_STATE(T_ACCUM_FRAC,
1567 P 1695 1 NUMBER_TRANSITION(DIGIT, DO NOTHING, ACCUM_FRAC),
1568 P 1696 1 NUMBER_TRANSITION(OTHER, NOT_NUMBER, END_STATE)),
1569 P 1697 1
1570 P 1698 1 NUMBER_STATE(GET_EXPONENT,
1571 P 1699 1 NUMBER_TRANSITION(DIGIT, DO NOTHING, ACCUM_EXP),
1572 P 1700 1 NUMBER_TRANSITION(PLUS, DO NOTHING, GET_EXP_SIGN),
1573 P 1701 1 NUMBER_TRANSITION(MINUS, DO NOTHING, GET_EXP_SIGN),
```



```
1574 P 1702 1 NUMBER_TRANSITION(OTHER, NOT_NUMBER, END_STATE)),
1575 P 1703 1
1576 P 1704 1 NUMBER_STATE(GET_EXP_SIGN,
1577 P 1705 1 NUMBER_TRANSITION(DIGIT, DO_NOTHING, ACCUM_EXP),
1578 P 1706 1 NUMBER_TRANSITION(OTHER, NOT_NUMBER, END_STATE)),
1579 P 1707 1
1580 P 1708 1 NUMBER_STATE(ACCUM_EXP,
1581 P 1709 1 NUMBER_TRANSITION(DIGIT, DO_NOTHING, ACCUM_EXP),
1582 P 1710 1 NUMBER_TRANSITION(UNDERSCORE, DO_NOTHING, T_ACCUM_EXP),
1583 P 1711 1 NUMBER_TRANSITION(OTHER, GOT_NUMBER, END_STATE)),
1584 P 1712 1
1585 P 1713 1 NUMBER_STATE(T_ACCUM_EXP,
1586 P 1714 1 NUMBER_TRANSITION(DIGIT, DO_NOTHING, ACCUM_EXP),
1587 P 1715 1 NUMBER_TRANSITION(OTHER, NOT_NUMBER, END_STATE)),
1588 P 1716 1
1589 P 1717 1 NUMBER_STATE(B_START_STATE,
1590 P 1718 1 NUMBER_TRANSITION(DIGIT, DO_NOTHING, B_ACCUM_INT),
1591 P 1719 1 NUMBER_TRANSITION(OTHER, NOT_NUMBER, END_STATE)),
1592 P 1720 1
1593 P 1721 1 NUMBER_STATE(B_ACCUM_INT,
1594 P 1722 1 NUMBER_TRANSITION(DIGIT, DO_NOTHING, B_ACCUM_INT),
1595 P 1723 1 NUMBER_TRANSITION(HEXDIGIT, DO_NOTHING, B_ACCUM_INT),
1596 P 1724 1 NUMBER_TRANSITION(B, DO_NOTHING, B_ACCUM_INT),
1597 P 1725 1 NUMBER_TRANSITION(D, DO_NOTHING, B_ACCUM_INT),
1598 P 1726 1 NUMBER_TRANSITION(E, DO_NOTHING, B_ACCUM_INT),
1599 P 1727 1 NUMBER_TRANSITION(UNDERSCORE, DO_NOTHING, T_B_ACCUM_INT),
1600 P 1728 1 NUMBER_TRANSITION(DOT, MARK_DEC_PT, B_ACCUM_FRAC),
1601 P 1729 1 NUMBER_TRANSITION(POUND, DO_NOTHING, GET_EXPONENT),
1602 P 1730 1 NUMBER_TRANSITION(OTHER, GOT_NUMBER, END_STATE)),
1603 P 1731 1
1604 P 1732 1 NUMBER_STATE(T_B_ACCUM_INT,
1605 P 1733 1 NUMBER_TRANSITION(DIGIT, DO_NOTHING, B_ACCUM_INT),
1606 P 1734 1 NUMBER_TRANSITION(HEXDIGIT, DO_NOTHING, B_ACCUM_INT),
1607 P 1735 1 NUMBER_TRANSITION(B, DO_NOTHING, B_ACCUM_INT),
1608 P 1736 1 NUMBER_TRANSITION(D, DO_NOTHING, B_ACCUM_INT),
1609 P 1737 1 NUMBER_TRANSITION(E, DO_NOTHING, B_ACCUM_INT),
1610 P 1738 1 NUMBER_TRANSITION(OTHER, NOT_NUMBER, END_STATE)),
1611 P 1739 1
1612 P 1740 1 NUMBER_STATE(B_ACCUM_FRAC,
1613 P 1741 1 NUMBER_TRANSITION(DIGIT, DO_NOTHING, B_ACCUM_FRAC),
1614 P 1742 1 NUMBER_TRANSITION(HEXDIGIT, DO_NOTHING, B_ACCUM_FRAC),
1615 P 1743 1 NUMBER_TRANSITION(B, DO_NOTHING, B_ACCUM_FRAC),
1616 P 1744 1 NUMBER_TRANSITION(D, DO_NOTHING, B_ACCUM_FRAC),
1617 P 1745 1 NUMBER_TRANSITION(E, DO_NOTHING, B_ACCUM_FRAC),
1618 P 1746 1 NUMBER_TRANSITION(UNDERSCORE, DO_NOTHING, T_B_ACCUM_FRAC),
1619 P 1747 1 NUMBER_TRANSITION(POUND, MARK_E_EXP, GET_EXPONENT),
1620 P 1748 1 NUMBER_TRANSITION(OTHER, NOT_NUMBER, END_STATE)),
1621 P 1749 1
1622 P 1750 1 NUMBER_STATE(T_B_ACCUM_FRAC,
1623 P 1751 1 NUMBER_TRANSITION(DIGIT, DO_NOTHING, B_ACCUM_FRAC),
1624 P 1752 1 NUMBER_TRANSITION(HEXDIGIT, DO_NOTHING, B_ACCUM_FRAC),
1625 P 1753 1 NUMBER_TRANSITION(B, DO_NOTHING, B_ACCUM_FRAC),
1626 P 1754 1 NUMBER_TRANSITION(D, DO_NOTHING, B_ACCUM_FRAC),
1627 P 1755 1 NUMBER_TRANSITION(E, DO_NOTHING, B_ACCUM_FRAC),
1628 P 1756 1 NUMBER_TRANSITION(OTHER, NOT_NUMBER, END_STATE)),
1629 P 1757 1
1630 P 1758 1 NUMBER_STATE(END_STATE,
```

```
1631 1759 1 NUMBER_TRANSITION(OTHER, GIVE_ERROR, END_STATE));
1632 1760 1
1633 1761 1
1634 1762 1 ! Save away the value of the B_START_STATE state for ADA. This will be
1635 1763 1 ! used later in the number scanner.
1636 1764 1
1637 1765 1 COMPILETIME
1638 1766 1 REMEMBER ADA_B_START_STATE = 0;
1639 1767 1 %ASSIGN (REMEMBER_ADA_B_START_STATE, NUMST$XX_STATE_B_START_STATE);
1640 1768 1
1641 1769 1
1642 1770 1 ! Define the Primary Parser State Table for language ADA. Each transition
1643 1771 1 ! Entry in the state table has this format:
1644 1772 1
1645 1773 1 PRIMARY_TRANSITION(operator-code, action, next-state)
1646 1774 1
1647 1775 1 where the first parameter is the operator code which causes the transition
1648 1776 1 to be taken, the second parameter is the action routine CASE index for the
1649 1777 1 transition, and the third parameter is the next state in the Finite-State
1650 1778 1 Machine.
1651 1779 1
1652 P 1780 1 PRIMARY_STATE_TABLE(ADA_PRIMARY_TABLE,
1653 P 1781 1
1654 P 1782 1 PRIMARY_STATE(START_STATE,
1655 P 1783 1 PRIMARY_TRANSITION(GLOBAL SLASH, START_GBL, GET_GLOBAL),
1656 P 1784 1 PRIMARY_TRANSITION(BACKSLASH, START_SLASH, GOT_BACKSLASH),
1657 P 1785 1 PRIMARY_TRANSITION(INVOCNUM, SLASH_INVOCNUM, GOT_BACKSLASH),
1658 P 1786 1 PRIMARY_TRANSITION(DOT, START_DOT, GOT_DOT),
1659 P 1787 1 PRIMARY_TRANSITION(SUBSCRIPT, START_SUBSCR, GOT_SUBSCRIPT),
1660 P 1788 1 PRIMARY_TRANSITION(ADA TICK, START_TICK, END_STATE),
1661 P 1789 1 PRIMARY_TRANSITION(PRIMARY_TERM, START_TERM, END_STATE)),
1662 P 1790 1
1663 P 1791 1 PRIMARY_STATE(GET GLOBAL,
1664 P 1792 1 PRIMARY_TRANSITION(PRIMARY_TERM, GBL_TERM, END_STATE)),
1665 P 1793 1
1666 P 1794 1 PRIMARY_STATE(GOT BACKSLASH,
1667 P 1795 1 PRIMARY_TRANSITION(BACKSLASH, SLASH SLASH, GOT_BACKSLASH),
1668 P 1796 1 PRIMARY_TRANSITION(INVOCNUM, SLASH_INVOCNUM, GOT_BACKSLASH),
1669 P 1797 1 PRIMARY_TRANSITION(DOT, SLASH DOT, GOT DOT),
1670 P 1798 1 PRIMARY_TRANSITION(SUBSCRIPT, SLASH SUBSCR, GOT SUBSCRIPT),
1671 P 1799 1 PRIMARY_TRANSITION(ADA TICK, SLASH TICK, END_STATE),
1672 P 1800 1 PRIMARY_TRANSITION(PRIMARY_TERM, SLASH_TERM, END_STATE)),
1673 P 1801 1
1674 P 1802 1 PRIMARY_STATE(GOT DOT,
1675 P 1803 1 PRIMARY_TRANSITION(DOT, DOT DOT, GOT DOT),
1676 P 1804 1 PRIMARY_TRANSITION(SUBSCRIPT, DOT SUBSCR, GOT SUBSCRIPT),
1677 P 1805 1 PRIMARY_TRANSITION(ADA TICK, DOT TICK, END_STATE),
1678 P 1806 1 PRIMARY_TRANSITION(PRIMARY_TERM, DOT_TERM, END_STATE)),
1679 P 1807 1
1680 P 1808 1 PRIMARY_STATE(GOT SUPSCRIPT,
1681 P 1809 1 PRIMARY_TRANSITION(DOT, SUBSCR DOT, GOT DOT),
1682 P 1810 1 PRIMARY_TRANSITION(SUBSCRIPT, SUBSCR SUBSCR, GOT SUBSCRIPT),
1683 P 1811 1 PRIMARY_TRANSITION(PRIMARY_TERM, SUBSCR_TERM, END_STATE)),
1684 P 1812 1
1685 1813 1 PRIMARY_STATE(END_STATE));
1686 1814 1
1687 1815 1
```

```

: 1688      1816 1  ! Define the table of pointers to the parse tables for ADA.
: 1689      1817 1
: 1690      P 1818 1 LANGUAGE_TABLES(LANGUAGE = ADA,
: 1691      P 1819 1     CHARTBL = ADA_CHARTBL,
: 1692      P 1820 1     IDENT_OPTBL = ADA_IDENT_OPTBL,
: 1693      P 1821 1     OPCHAR_OPTBL = ADA_OPCHAR_OPTBL,
: 1694      P 1822 1     NUMBER_TABLE = ADA_NUMBER_TABLE,
: 1695      P 1823 1     PRIMARY_TABLE = ADA_PRIMARY_TABLE,
: 1696      P 1824 1     SUBSCR_TERMS = ADA_SUBSCR_TERM_TBL,
: 1697      P 1825 1     PRIDTBL = ADA_PRID_TABLE,
: 1698      1826 1     BIF_TABLE = ADA_FUNCTION_TABLE);

```


BASIC PARSE TABLES

This section includes all the Lexical Scanner and Parser tables needed to scan and parse BASIC expressions.

Define the BASIC Character Table. What is listed here is actually a list of exceptions to the Character Table for language UNKNOWN. (Language UNKNOWN lists the "average" use of each character in the character set.)

```
CHAR_EXCEPTION TABLE(BASIC CHARTBL,  
  CHAR_ENTRY(':', DOT, NUMBER_START, IDENT_MIDDLE, IDENT_END, ADDRESS_OP, SPECIAL_SYMBOL),  
  CHAR_ENTRY('%', OTHER, IDENT_END),  
  CHAR_ENTRY(':', OTHER, OPCHAR, OPCHAR_INFIX, TERMINATOR),  
  CHAR_ENTRY('.', OTHER, OPCHAR, SPECIAL_SYMBOL),  
  CHAR_ENTRY('<', OTHER, OPCHAR, OPCHAR_INFIX, ADDRESS_OP, TERMINATOR),  
  CHAR_ENTRY('>', OTHER, OPCHAR, OPCHAR_INFIX, TERMINATOR),  
  CHAR_ENTRY('=', OTHER, OPCHAR, OPCHAR_INFIX, TERMINATOR));
```

Define the BASIC Operator Table for operators whose names are identifiers.

```
OPERATOR TABLE(BASIC IDENT_OPTBL,  
  OPERATOR_ENTRY('NOT', BIT_NOT, PREFIX, 200, 45),  
  OPERATOR_ENTRY('AND', BIT_AND, INFIX, 40, 40),  
  OPERATOR_ENTRY('OR', BIT_OR, INFIX, 30, 30),  
  OPERATOR_ENTRY('XOR', BIT_XOR, INFIX, 30, 30),  
  OPERATOR_ENTRY('IMP', BIT_IMP, INFIX, 20, 20),  
  OPERATOR_ENTRY('EQV', BIT_EQV, INFIX, 10, 10));
```

Define the BASIC Operator Table for operators whose names are composed of operator characters such as "++" or "++". This table includes those operators which are part of DEBUG Primary Symbols (such as "\").

```
OPERATOR TABLE(BASIC OPCHAR_OPTBL,  
  OPERATOR_ENTRY('\', GLOBAL_SLASH, PREFIX, 0, 0, PRIMARY),  
  OPERATOR_ENTRY('\', BACKSLASH, INFIX, 0, 0, PRIMARY),  
  OPERATOR_ENTRY(':', DOT, INFIX, 0, 0, PRIMARY),  
  OPERATOR_ENTRY('(', SUBSCRIPT, POSTFIX, 0, 0, PRIMARY),  
  OPERATOR_ENTRY('(', OPENPAREN, PREFIX, 200, 5, LEXICAL),  
  OPERATOR_ENTRY(')', CLOSEPAREN, POSTFIX, 6, 200, LEXICAL),  
  OPERATOR_ENTRY('+', UNARY_PLUS, PREFIX, 200, 70),  
  OPERATOR_ENTRY('-', UNARY_MINUS, PREFIX, 200, 70),  
  OPERATOR_ENTRY('^', POWER_OF, INFIX, 92, 90),  
  OPERATOR_ENTRY('^', POWER_OF, INFIX, 92, 90),  
  OPERATOR_ENTRY('*', MULTIPLY, INFIX, 80, 80),  
  OPERATOR_ENTRY('/', DIVIDE, INFIX, 80, 80),  
  OPERATOR_ENTRY('+', ADD, INFIX, 60, 60),  
  OPERATOR_ENTRY('-', SUBTRACT, INFIX, 60, 60),  
  OPERATOR_ENTRY('<', LSS_THAN, INFIX, 50, 50),  
  OPERATOR_ENTRY('<=', LSS_EQUAL, INFIX, 50, 50),  
  OPERATOR_ENTRY('=<', LSS_EQUAL, INFIX, 50, 50),
```

```
1757 P 1884 1 OPERATOR_ENTRY('>', GTR_THAN, INFIX, 50, 50),
1758 P 1885 1 OPERATOR_ENTRY('>=', GTR_EQUAL, INFIX, 50, 50),
1759 P 1886 1 OPERATOR_ENTRY('=>', GTR_EQUAL, INFIX, 50, 50),
1760 P 1887 1 OPERATOR_ENTRY('=', EQUAL, INFIX, 50, 50),
1761 P 1888 1 OPERATOR_ENTRY('<>', NOT_EQUAL, INFIX, 50, 50),
1762 1889 1 OPERATOR_ENTRY('><', NOT_EQUAL, INFIX, 50, 50));
1763 1890 1
1764 1891 1
1765 1892 1 ! Define the BASIC Terminator Lexical Token Table for subscript expressions.
1766 1893 1 ! In BASIC a subscript expression can be terminated by ")" (end of sub-
1767 1894 1 ! scripts), by "," (more subscripts to follow), or by ":" (string subscript
1768 1895 1 ! upper bound to follow).
1769 1896 1
1770 P 1897 1 TERMINATOR_TABLE(BASIC_SUBSCR_TERM_TBL,
1771 P 1898 1 TERMINATOR_ENTRY(')', TERM_CLOSE, BALANCED_PARENS),
1772 P 1899 1 TERMINATOR_ENTRY(':', TERM_COLON, MUST_BE_SINGLE),
1773 1900 1 TERMINATOR_ENTRY(',', TERM_COMMA));
1774 1901 1
1775 1902 1
1776 1903 1 ! Define the BASIC Predefined Identifier Table.
1777 1904 1
1778 1905 1 PRID_TABLE(BASIC_PRID_TABLE);
1779 1906 1
1780 1907 1
1781 1908 1 ! Define the BASIC Built-in Function Table.
1782 1909 1
1783 1910 1 BUILT_IN_FUNCTION_TABLE(BASIC_FUNCTION_TABLE);
1784 1911 1
1785 1912 1
1786 1913 1 ! Define the BASIC Number Scanner State Table. This table defines the states
1787 1914 1 ! of a Finite-State Machine which picks up all valid numeric constants in the
1788 1915 1 ! language.
1789 1916 1
1790 1917 1 ! The BASIC number table is the same as the UNKNOWN number table at present.
1791 1918 1
1792 1919 1 BIND
1793 1920 1 BASIC_NUMBER_TABLE = UNKNOWN_NUMBER_TABLE;
1794 1921 1
1795 1922 1
1796 1923 1 ! Define the Primary Parser State Table for language BASIC Each Transition
1797 1924 1 ! Entry in the state table has this format:
1798 1925 1
1799 1926 1 PRIMARY_TRANSITION(operator-code, action, next-state)
1800 1927 1
1801 1928 1 ! where the first parameter is the operator code which causes the transition
1802 1929 1 ! to be taken, the second parameter is the action routine CASE index for the
1803 1930 1 ! transition, and the third parameter is the next state in the Finite-State
1804 1931 1 ! Machine.
1805 1932 1
1806 P 1933 1 PRIMARY_STATE_TABLE(BASIC_PRIMARY_TABLE,
1807 P 1934 1
1808 P 1935 1 PRIMARY_STATE(START_STATE,
1809 P 1936 1 PRIMARY_TRANSITION(GLOBAL_SLASH, START_GBL, GET_GLOBAL),
1810 P 1937 1 PRIMARY_TRANSITION(BACKSLASH, START_SLASH, GOT_BACKSLASH),
1811 P 1938 1 PRIMARY_TRANSITION(INVOCNUM, SLASH_INVOCNUM, GOT_BACKSLASH),
1812 P 1939 1 PRIMARY_TRANSITION(DOT, START_DOT, GOT_DOT),
1813 P 1940 1 PRIMARY_TRANSITION(SUBSCRIPT, START_SUBSCR, GOT_SUBSCRIPT),
```

```
1814 P 1941 1 PRIMARY_TRANSITION(PRIMARY_TERM, START_TERM, END_STATE)),
1815 P 1942 1
1816 P 1943 1 PRIMARY_STATE(GET GLOBAL,
1817 P 1944 1 PRIMARY_TRANSITION(PRIMARY_TERM, GBL_TERM, END_STATE)),
1818 P 1945 1
1819 P 1946 1 PRIMARY_STATE(GOT BACKSLASH,
1820 P 1947 1 PRIMARY_TRANSITION(BACKSLASH, SLASH SLASH, GOT BACKSLASH),
1821 P 1948 1 PRIMARY_TRANSITION(INVOCNUM, SLASH_INVOCNUM, GOT BACKSLASH),
1822 P 1949 1 PRIMARY_TRANSITION(DOT, SLASH_DOT, GOT DOT),
1823 P 1950 1 PRIMARY_TRANSITION(SUBSCRIPT, SLASH SUBSCR, GOT SUBSCRIPT),
1824 P 1951 1 PRIMARY_TRANSITION(PRIMARY_TERM, SLASH_TERM, END_STATE)),
1825 P 1952 1
1826 P 1953 1 PRIMARY_STATE(GOT DOT,
1827 P 1954 1 PRIMARY_TRANSITION(DOT, DOT DOT, GOT DOT),
1828 P 1955 1 PRIMARY_TRANSITION(SUBSCRIPT, DOT SUBSCR, GOT SUBSCRIPT),
1829 P 1956 1 PRIMARY_TRANSITION(PRIMARY_TERM, DOT_TERM, END_STATE)),
1830 P 1957 1
1831 P 1958 1 PRIMARY_STATE(GOT SUBSCRIPT,
1832 P 1959 1 PRIMARY_TRANSITION(DOT, SUBSCR DOT, GOT DOT),
1833 P 1960 1 PRIMARY_TRANSITION(SUBSCRIPT, SUBSCR SUBSCR, GOT SUBSCRIPT),
1834 P 1961 1 PRIMARY_TRANSITION(PRIMARY_TERM, SUBSCR_TERM, END_STATE)),
1835 P 1962 1
1836 1963 1 PRIMARY_STATE(END_STATE));
1837 1964 1
1838 1965 1
1839 1966 1 ! Define the table of pointers to the parse tables for BASIC.
1840 1967 1 !
1841 P 1968 1 LANGUAGE_TABLES(LANGUAGE = BASIC,
1842 P 1969 1 CHARTBL = BASIC CHARTBL,
1843 P 1970 1 IDENT_OPTBL = BASIC IDENT_OPTBL,
1844 P 1971 1 OPCHAR_OPTBL = BASIC_OPCHAR_OPTBL,
1845 P 1972 1 NUMBER_TABLE = BASIC_NUMBER_TABLE,
1846 P 1973 1 PRIMARY_TABLE = BASIC_PRIMARY_TABLE,
1847 P 1974 1 SUBSCR_TERMS = BASIC_SUBSCR_TERM_TBL,
1848 P 1975 1 PRIDTBL = BASIC_PRID_TABLE,
1849 P 1976 1 BIF_TABLE = BASIC_FUNCTION_TABLE,
1850 1977 1 INCOMPLETE_QUAL = TRUE);
```


BLISS PARSE TABLES

This section includes all the Lexical Scanner and Parser tables needed to scan and parse the BLISS language.

Define the BLISS Character Table. What is listed here is actually a list of exceptions to the Character Table for Language UNKNOWN.

```
CHAR_EXCEPTION_TABLE(BLISS_CHAR_TBL,  
  CHAR_ENTRY('$', OTHER, IDENT_ANYWHERE),  
  CHAR_ENTRY('<', OTHER, OPCHAR, ADDRESS_OP),  
  CHAR_ENTRY('>', OTHER, TERMINATOR),  
  CHAR_ENTRY('[', OTHER, OPCHAR),  
  CHAR_ENTRY(']', OTHER, TERMINATOR),  
  CHAR_ENTRY('^', OTHER, OPCHAR, SPECIAL_SYMBOL),  
  CHAR_ENTRY('_', OTHER, IDENT_ANYWHERE));
```

Define the BLISS Operator Table for operators whose names are identifiers.

```
OPERATOR_TABLE(BLISS_IDENT_OPTBL,  
  OPERATOR_ENTRY('MOD', REMAINDER, INFIX, 70, 70),  
  OPERATOR_ENTRY('EQL', EQUAL, INFIX, 50, 50),  
  OPERATOR_ENTRY('EQLU', EQUAL, INFIX, 50, 50),  
  OPERATOR_ENTRY('EQLA', EQUAL, INFIX, 50, 50),  
  OPERATOR_ENTRY('NEQ', NOT_EQUAL, INFIX, 50, 50),  
  OPERATOR_ENTRY('NEQU', NOT_EQUAL, INFIX, 50, 50),  
  OPERATOR_ENTRY('NEQA', NOT_EQUAL, INFIX, 50, 50),  
  OPERATOR_ENTRY('GTR', GTR_THAN, INFIX, 50, 50),  
  OPERATOR_ENTRY('GTRU', GTR_THAN_U, INFIX, 50, 50),  
  OPERATOR_ENTRY('GTRA', GTR_THAN_U, INFIX, 50, 50),  
  OPERATOR_ENTRY('GEQ', GTR_EQUAL, INFIX, 50, 50),  
  OPERATOR_ENTRY('GEQU', GTR_EQUAL_U, INFIX, 50, 50),  
  OPERATOR_ENTRY('GEQA', GTR_EQUAL_U, INFIX, 50, 50),  
  OPERATOR_ENTRY('LSS', LSS_THAN, INFIX, 50, 50),  
  OPERATOR_ENTRY('LSSU', LSS_THAN_U, INFIX, 50, 50),  
  OPERATOR_ENTRY('LSSA', LSS_THAN_U, INFIX, 50, 50),  
  OPERATOR_ENTRY('LEQ', LSS_EQUAL, INFIX, 50, 50),  
  OPERATOR_ENTRY('LEQU', LSS_EQUAL_U, INFIX, 50, 50),  
  OPERATOR_ENTRY('LEQA', LSS_EQUAL_U, INFIX, 50, 50),  
  OPERATOR_ENTRY('NOT', BIT_NOT, PREFIX, 200, 40),  
  OPERATOR_ENTRY('AND', BIT_AND, INFIX, 30, 30),  
  OPERATOR_ENTRY('OR', BIT_OR, INFIX, 20, 20),  
  OPERATOR_ENTRY('EQV', BIT_EQV, INFIX, 10, 10),  
  OPERATOR_ENTRY('XOR', BIT_XOR, INFIX, 10, 10));
```

Define the BLISS Operator Table for operators whose names are composed of operator characters such as '+', '-', or '*'. This table includes operators which are part of DEBUG Primary Symbols (such as '\').

```
OPERATOR_TABLE(BLISS_OPCHAR_OPTBL,  
  OPERATOR_ENTRY('\ ', GLOBAL_SLASH, PREFIX, 0, 0, PRIMARY),  
  OPERATOR_ENTRY('\ ', BACKSLASH, INFIX, 0, 0, PRIMARY),
```

```
1909 P 2035 1 OPERATOR_ENTRY('[' , SUBSCRIPT, POSTFIX, 0, 0, PRIMARY),
1910 P 2036 1
1911 P 2037 1 OPERATOR_ENTRY('(' , OPENPAREN, PREFIX, 200, 5, LEXICAL),
1912 P 2038 1 OPERATOR_ENTRY(')' , CLOSEPAREN, POSTFIX, 6, 200, LEXICAL),
1913 P 2039 1
1914 P 2040 1 OPERATOR_ENTRY('<' , BIT_SELECT, POSTFIX, 110, 200, LEXICAL),
1915 P 2041 1 OPERATOR_ENTRY('.') , INDIRECT, PREFIX, 200, 100),
1916 P 2042 1 OPERATOR_ENTRY('+.' , UNARY_PLUS, PREFIX, 200, 90),
1917 P 2043 1 OPERATOR_ENTRY('-' , UNARY_MINUS, PREFIX, 200, 90),
1918 P 2044 1 OPERATOR_ENTRY('^.' , LEFT_SHIFT, INFIX, 80, 80),
1919 P 2045 1 OPERATOR_ENTRY('*.' , MULTIPLY, INFIX, 70, 70),
1920 P 2046 1 OPERATOR_ENTRY('/' , DIVIDE, INFIX, 70, 70),
1921 P 2047 1 OPERATOR_ENTRY('+' , ADD, INFIX, 60, 60),
1922 P 2048 1 OPERATOR_ENTRY('-' , SUBTRACT, INFIX, 60, 60),
1923 P 2049 1
1924 P 2050 1
1925 P 2051 1 ! Define the BLISS Terminator Lexical Token Table for subscript expressions.
1926 P 2052 1
1927 P 2053 1 TERMINATOR_TABLE(BLISS_SUBSCR_TERM_TBL,
1928 P 2054 1 TERMINATOR_ENTRY(']' , TERM_CLOSE),
1929 P 2055 1 TERMINATOR_ENTRY(':', TERM_COLON, MUST_BE_SINGLE),
1930 P 2056 1 TERMINATOR_ENTRY(',') , TERM_COMMA));
1931 P 2057 1
1932 P 2058 1
1933 P 2059 1 ! Define the BLISS Predefined Identifier Table.
1934 P 2060 1
1935 P 2061 1 PRID_TABLE(BLISS_PRID_TABLE);
1936 P 2062 1
1937 P 2063 1
1938 P 2064 1 ! Define the BLISS Built-in Function Table.
1939 P 2065 1
1940 P 2066 1 BUILT_IN_FUNCTION_TABLE(BLISS_FUNCTION_TABLE);
1941 P 2067 1
1942 P 2068 1
1943 P 2069 1 ! Define the BLISS Number Scanner State Table. This table defines the states
1944 P 2070 1 of a Finite-State Machine which picks up all valid numeric constants in the
1945 P 2071 1 language.
1946 P 2072 1
1947 P 2073 1 The BLISS number table is the same as the UNKNOWN number table at present.
1948 P 2074 1
1949 P 2075 1 BIND
1950 P 2076 1 BLISS_NUMBER_TABLE = UNKNOWN_NUMBER_TABLE;
1951 P 2077 1
1952 P 2078 1
1953 P 2079 1 ! Define the Primary Parser State Table for language BLISS. Each transition
1954 P 2080 1 Entry in the state table has this format:
1955 P 2081 1
1956 P 2082 1 PRIMARY_TRANSITION(operator-code, action, next-state)
1957 P 2083 1
1958 P 2084 1 where the first parameter is the operator code which causes the transition
1959 P 2085 1 to be taken, the second parameter is the action routine CASE index for the
1960 P 2086 1 transition, and the third parameter is the next state in the Finite-State
1961 P 2087 1 Machine.
1962 P 2088 1
1963 P 2089 1 PRIMARY_STATE_TABLE(BLISS_PRIMARY_TABLE,
1964 P 2090 1
1965 P 2091 1 PRIMARY_STATE(START_STATE,
```

```
1966 P 2092 1 PRIMARY_TRANSITION(GLOBAL SLASH, START GBL, GET GLOBAL),
1967 P 2093 1 PRIMARY_TRANSITION(BACKSLASH, START SLASH, GOT BACKSLASH),
1968 P 2094 1 PRIMARY_TRANSITION(INVOCNUM, SLASH INVOCNUM, GOT BACKSLASH),
1969 P 2095 1 PRIMARY_TRANSITION(SUBSCRIPT, START SUBSCR BLI, GOT SUBSCRIPT),
1970 P 2096 1 PRIMARY_TRANSITION(PRIMARY_TERM, START_TERM, END_STATE)),
1971 P 2097 1
1972 P 2098 1 PRIMARY_STATE(GET GLOBAL,
1973 P 2099 1 PRIMARY_TRANSITION(PRIMARY_TERM, GBL_TERM, END_STATE)),
1974 P 2100 1
1975 P 2101 1 PRIMARY_STATE(GOT BACKSLASH,
1976 P 2102 1 PRIMARY_TRANSITION(BACKSLASH, SLASH SLASH, GOT BACKSLASH),
1977 P 2103 1 PRIMARY_TRANSITION(INVOCNUM, SLASH INVOCNUM, GOT BACKSLASH),
1978 P 2104 1 PRIMARY_TRANSITION(SUBSCRIPT, SLASH SUBSCR BLI, GOT SUBSCRIPT),
1979 P 2105 1 PRIMARY_TRANSITION(PRIMARY_TERM, SLASH_TERM, END_STATE)),
1980 P 2106 1
1981 P 2107 1 PRIMARY_STATE(GOT SUBSCRIPT,
1982 P 2108 1 PRIMARY_TRANSITION(PRIMARY_TERM, SUBSCR_TERM, END_STATE)),
1983 P 2109 1
1984 2110 1 PRIMARY_STATE(END_STATE));
1985 2111 1
1986 2112 1
1987 2113 1 ! Define the table of pointers to the parse tables for BLISS.
1988 2114 1 !
1989 P 2115 1 LANGUAGE_TABLES(LANGUAGE = BLISS,
1990 P 2116 1 CHARTBL = BLISS_CHARTBL,
1991 P 2117 1 IDENT_OPTBL = BLISS_IDENT_OPTBL,
1992 P 2118 1 OPCHAR_OPTBL = BLISS_OPCHAR_OPTBL,
1993 P 2119 1 NUMBER_TABLE = BLISS_NUMBER_TABLE,
1994 P 2120 1 PRIMARY_TABLE = BLISS_PRIMARY_TABLE,
1995 P 2121 1 SUBSCR_TERMS = BLISS_SUBSCR_TERM_TBL,
1996 P 2122 1 PRIDTBL = BLISS_PRID_TABLE,
1997 2123 1 BIF_TABLE = BLISS_FUNCTION_TABLE);
```


C P A R S E T A B L E S

This section includes all the Lexical Scanner and Parser tables needed to scan and parse the C language.

Define the C Character Table. What is listed here is actually a list of exceptions to the Character Table for Language UNKNOWN.

```
CHAR_EXCEPTION_TABLE(C_CHARTBL,  
  CHAR_ENTRY('$', OTHER, IDENT_ANYWHERE),  
  CHAR_ENTRY('.', OTHER, IDENT_ANYWHERE),  
  CHAR_ENTRY('t', OTHER, OPCHAR, OPCHAR_INFIX),  
  CHAR_ENTRY('*', OTHER, OPCHAR, ADDRESS_OP),  
  CHAR_ENTRY('<', OTHER, OPCHAR),  
  CHAR_ENTRY('>', OTHER, OPCHAR, OPCHAR_INFIX, ADDRESS_OP),  
  CHAR_ENTRY('=', OTHER, OPCHAR, OPCHAR_INFIX, TERMINATOR),  
  CHAR_ENTRY('>', OTHER, OPCHAR, OPCHAR_INFIX, TERMINATOR),  
  CHAR_ENTRY('[', OTHER, OPCHAR),  
  CHAR_ENTRY(']', OTHER, TERMINATOR),  
  CHAR_ENTRY('^', OTHER, OPCHAR, SPECIAL_SYMBOL),  
  CHAR_ENTRY('!', OTHER, OPCHAR, OPCHAR_INFIX),  
  CHAR_ENTRY('-', OTHER, OPCHAR));
```

Define the C Operator Table for operators whose names are identifiers.

```
OPERATOR_TABLE(C_IDENT_OPTBL,  
  OPERATOR_ENTRY('sizeof', sizeof, PREFIX, 200, 140));
```

Define the C Operator Table for operators whose names are composed of operator characters such as '+', '-', or '*'. This table includes operators which are part of DEBUG Primary Symbols (such as '\').

```
OPERATOR_TABLE(C_OPCHAR_OPTBL,  
  OPERATOR_ENTRY('\', GLOBAL_SLASH, PREFIX, 0, 0, PRIMARY),  
  OPERATOR_ENTRY('\', BACKSLASH, INFIX, 0, 0, PRIMARY),  
  OPERATOR_ENTRY('[', SUBSCRIPT, POSTFIX, 0, 0, PRIMARY),  
  OPERATOR_ENTRY('.', DOT, INFIX, 0, 0, PRIMARY),  
  
  OPERATOR_ENTRY('(', OPENPAREN, PREFIX, 200, 5, LEXICAL),  
  OPERATOR_ENTRY(')', CLOSEPAREN, POSTFIX, 6, 200, LEXICAL),  
  OPERATOR_ENTRY('!', NOT, PREFIX, 200, 140),  
  OPERATOR_ENTRY('~', BIT_NOT, PREFIX, 200, 140),  
  OPERATOR_ENTRY('*', INDIRECT, PREFIX, 200, 140),  
  OPERATOR_ENTRY('*', MULTIPLY, INFIX, 130, 130),  
  OPERATOR_ENTRY('/', DIVIDE, INFIX, 130, 130),  
  OPERATOR_ENTRY('%', REMAINDER, INFIX, 130, 130),  
  OPERATOR_ENTRY('<<', LEFT_SHIFT, INFIX, 110, 110),  
  OPERATOR_ENTRY('>>', RIGHT_SHIFT, INFIX, 110, 110),  
  OPERATOR_ENTRY('<', LSS_THAN, INFIX, 100, 100),  
  OPERATOR_ENTRY('<=', LSS_EQUAL, INFIX, 100, 100),  
  OPERATOR_ENTRY('>', GTR_THAN, INFIX, 100, 100),
```

```
2056 P 2131 1 OPERATOR_ENTRY('>=', GTR_EQUAL, INFIX, 100, 100),
2057 P 2182 1 OPERATOR_ENTRY('==', EQUAL, INFIX, 90, 90),
2058 P 2183 1 OPERATOR_ENTRY('!=', NOT_EQUAL, INFIX, 90, 90),
2059 P 2184 1 OPERATOR_ENTRY('&', BIT_XOR, INFIX, 70, 70),
2060 P 2185 1 OPERATOR_ENTRY('||', BIT_OR, INFIX, 60, 60),
2061 P 2186 1 OPERATOR_ENTRY('|||', SHORT_OR, INFIX, 40, 40));
2062
2063
2064 2187 1
2065 2188 1
2066 2189 1 ! Define Lexical Token Entries which require special-case scanning.
2067 2190 1
2068 2191 1 BIND
2069 2192 1 C_ADDR OF TOKEN =
2070 2193 1 OPERATOR_ENTRY('&', ADDRESS_OF, PREFIX, 200, 140),
2071 2194 1 C_BIT AND TOKEN =
2072 2195 1 OPERATOR_ENTRY('&', BIT_AND, INFIX, 80, 80),
2073 2196 1 C_AND TOKEN =
2074 2197 1 OPERATOR_ENTRY('&&', SHORT_AND, INFIX, 50, 50),
2075 2198 1 C_ADD TOKEN =
2076 2199 1 OPERATOR_ENTRY('+', ADD, INFIX, 120, 120),
2077 2200 1 C_MINUS TOKEN =
2078 2201 1 OPERATOR_ENTRY('-', UNARY_MINUS, PREFIX, 200, 140),
2079 2202 1 C_SUB TOKEN =
2080 2203 1 OPERATOR_ENTRY('-', SUBTRACT, INFIX, 120, 120),
2081 2204 1 C_ARROW TOKEN =
2082 2205 1 OPERATOR_ENTRY('->', PLI_DEREF, INFIX, 0, 0, PRIMARY),
2083 2206 1
2084 2207 1 ! The indirect operator will also be allowed as an address expression
2085 2208 1 operator in C (synonymous with '*' and '@'). That's why it appears
2086 2209 1 here as a special case, in addition to its appearance in the normal
2087 2210 1 operator tables. The precedence of '40' here is relative to other
2088 2211 1 address expression operators.
2089 2212 1
2090 2213 1 C_INDIRECT TOKEN =
2091 2214 1 OPERATOR_ENTRY('*', INDIRECT, PREFIX, 200, 40);
2092 2215 1
2093 2216 1
2094 2217 1 ! Increment and decrement were commented out so as not to allow operators with
2095 2218 1 side effects. If we decide to allow them, the comments can be removed.
2096 2219 1
2097 2220 1 BIND
2098 2221 1 C_PRE INCR TOKEN =
2099 2222 1 OPERATOR_ENTRY('++', PRE_INCR, PREFIX, 200, 140),
2100 2223 1 C_POST INCR TOKEN =
2101 2224 1 OPERATOR_ENTRY('++', POST_INCR, POSTFIX, 140, 200),
2102 2225 1 C_PRE DECR TOKEN =
2103 2226 1 OPERATOR_ENTRY('--', PRE_DECR, PREFIX, 200, 140),
2104 2227 1 C_POST DECR TOKEN =
2105 2228 1 OPERATOR_ENTRY('--', POST_DECR, POSTFIX, 140, 200);
2106 2229 1
2107 2230 1
2108 2231 1 ! Define the C Terminator Lexical Token Table for subscript expressions.
2109 2232 1
2110 2233 1 P 2234 1 TERMINATOR_TABLE(C_SUBSCR_TERM_TBL,
2111 2235 1 TERMINATOR_ENTRY(']', TERM_CLOSE),
2112 2236 1 TERMINATOR_ENTRY(':', TERM_COLON, MUST_BE_SINGLE),
2112 2237 1 TERMINATOR_ENTRY(',', TERM_COMMA));
```

```
2113 2238 1
2114 2239 1
2115 2240 1  ! Define the C Predefined Identifier Table.
2116 2241 1
2117 2242 1 PRID_TABLE(C_PRID_TABLE);
2118 2243 1
2119 2244 1
2120 2245 1  ! Define the C Built-in Function Table.
2121 2246 1
2122 2247 1 BUILT_IN_FUNCTION_TABLE(C_FUNCTION_TABLE);
2123 2248 1
2124 2249 1
2125 2250 1  ! Define the C Number Scanner State Table. This table defines the states
2126 2251 1  of a Finite-State Machine which picks up all valid numeric constants in the
2127 2252 1  language.
2128 2253 1
2129 2254 1  The C number table is the same as the UNKNOWN number table.
2130 2255 1
2131 2256 1 BIND
2132 2257 1  C_NUMBER_TABLE = UNKNOWN_NUMBER_TABLE;
2133 2258 1
2134 2259 1
2135 2260 1  ! Define the Primary Parser State Table for language C. Each transition
2136 2261 1  Entry in the state table has this format:
2137 2262 1
2138 2263 1  PRIMARY_TRANSITION(operator-code, action, next-state)
2139 2264 1
2140 2265 1  where the first parameter is the operator code which causes the transition
2141 2266 1  to be taken, the second parameter is the action routine CASE index for the
2142 2267 1  transition, and the third parameter is the next state in the Finite-State
2143 2268 1  Machine.
2144 2269 1
2145 P 2270 1 PRIMARY_STATE_TABLE(C_PRIMARY_TABLE,
2146 P 2271 1
2147 P 2272 1  PRIMARY_STATE(START_STATE,
2148 P 2273 1  PRIMARY_TRANSITION(GLOBAL_SLASH, START_GBL, GET_GLOBAL),
2149 P 2274 1  PRIMARY_TRANSITION(BACKSLASH, START_SLASH, GOT_BACKSLASH),
2150 P 2275 1  PRIMARY_TRANSITION(INVOCNUM, SLASH_INVOCNUM, GOT_BACKSLASH),
2151 P 2276 1  PRIMARY_TRANSITION(DOT, START_DOT, GOT_DOT),
2152 P 2277 1  PRIMARY_TRANSITION(SUBSCRIPT, START_SUBSCR, GOT_SUBSCRIPT),
2153 P 2278 1  PRIMARY_TRANSITION(PLI_DEREF, START_DEREF, GOT_DOT),
2154 P 2279 1  PRIMARY_TRANSITION(PRIMARY_TERM, START_TERM, END_STATE)),
2155 P 2280 1
2156 P 2281 1  PRIMARY_STATE(GET_GLOBAL,
2157 P 2282 1  PRIMARY_TRANSITION(PRIMARY_TERM, GBL_TERM, END_STATE)),
2158 P 2283 1
2159 P 2284 1  PRIMARY_STATE(GOT_BACKSLASH,
2160 P 2285 1  PRIMARY_TRANSITION(BACKSLASH, SLASH_SLASH, GOT_BACKSLASH),
2161 P 2286 1  PRIMARY_TRANSITION(INVOCNUM, SLASH_INVOCNUM, GOT_BACKSLASH),
2162 P 2287 1  PRIMARY_TRANSITION(DOT, SLASH_DOT, GOT_DOT),
2163 P 2288 1  PRIMARY_TRANSITION(SUBSCRIPT, SLASH_SUBSCR, GOT_SUBSCRIPT),
2164 P 2289 1  PRIMARY_TRANSITION(PLI_DEREF, SLASH_DEREF, GOT_DOT),
2165 P 2290 1  PRIMARY_TRANSITION(PRIMARY_TERM, SLASH_TERM, END_STATE)),
2166 P 2291 1
2167 P 2292 1  PRIMARY_STATE(GOT_DOT,
2168 P 2293 1  PRIMARY_TRANSITION(DOT, DOT_DOT, GOT_DOT),
2169 P 2294 1  PRIMARY_TRANSITION(SUBSCRIPT, DOT_SUBSCR, GOT_SUBSCRIPT),
```



```
2170 P 2295 1 PRIMARY_TRANSITION(PLI Deref, DOT Deref, GOT DOT),
2171 P 2296 1 PRIMARY_TRANSITION(PRIMARY_TERM, DOT_TERM, END_STATE)),
2172 P 2297 1
2173 P 2298 1 PRIMARY_STATE(GOT SUBSCRIPT,
2174 P 2299 1 PRIMARY_TRANSITION(DOT, SUBSCR DOT, GOT DOT),
2175 P 2300 1 PRIMARY_TRANSITION(SUBSCRIPT, SUBSCR SUBSCR, GOT SUBSCRIPT),
2176 P 2301 1 PRIMARY_TRANSITION(PLI Deref, SUBSCR Deref, GOT DOT),
2177 P 2302 1 PRIMARY_TRANSITION(PRIMARY_TERM, SUBSCR_TERM, END_STATE)),
2178 P 2303 1
2179 2304 1 PRIMARY_STATE(END_STATE));
2180 2305 1
2181 2306 1
2182 2307 1 ! Save away the value of the GOT_SUBSCRIPT state for C. This will be
2183 2308 1 ! used later in the expression parser.
2184 2309 1
2185 2310 1 COMPILETIME
2186 2311 1 REMEMBER C STATE GOT SUBSCRIPT = 0;
2187 2312 1 XASSIGN (REMEMBER_C_STATE_GOT_SUBSCRIPT, PRIMARY$XX_STATE_GOT_SUBSCRIPT);
2188 2313 1
2189 2314 1
2190 2315 1 ! Define the table of pointers to the parse tables for C.
2191 2316 1
2192 P 2317 1 LANGUAGE_TABLES(LANGUAGE = C,
2193 P 2318 1 CHARTBL = C_CHARTBL,
2194 P 2319 1 IDENT_OPTBL = C_IDENT_OPTBL,
2195 P 2320 1 OPCHAR_OPTBL = C_OPCHAR_OPTBL,
2196 P 2321 1 NUMBER_TABLE = C_NUMBER_TABLE,
2197 P 2322 1 PRIMARY_TABLE = C_PRIMARY_TABLE,
2198 P 2323 1 SUBSCR_TERMS = C_SUBSCR_TERM_TBL,
2199 P 2324 1 PRIDTBL = C_PRID_TABLE,
2200 P 2325 1 BIF_TABLE = C_FUNCTION_TABLE,
2201 P 2326 1 MULTIPLE_SUBSCR = TRUE,
2202 P 2327 1 ENFORCE_RECORD = FALSE,
2203 2328 1 CASING_SIGNIFICANT = TRUE);
```

COBOL PARSE TABLES

This section includes all the Lexical Scanner and Parser tables needed to scan and parse the COBOL language.

Define the COBOL Character Table. What is listed here is actually a list of exceptions to the Character Table for Language UNKNOWN.

```
CHAR_EXCEPTION TABLE(COBOL CHARTBL,  
  CHAR_ENTRY('-', MINUS, OPCHAR, ADDRESS_OP, IDENT_MIDDLE),  
  CHAR_ENTRY('0', DIGIT, DIGIT, IDENT_ANYWHERE, NUMBER_START),  
  CHAR_ENTRY('1', DIGIT, DIGIT, IDENT_ANYWHERE, NUMBER_START),  
  CHAR_ENTRY('2', DIGIT, DIGIT, IDENT_ANYWHERE, NUMBER_START),  
  CHAR_ENTRY('3', DIGIT, DIGIT, IDENT_ANYWHERE, NUMBER_START),  
  CHAR_ENTRY('4', DIGIT, DIGIT, IDENT_ANYWHERE, NUMBER_START),  
  CHAR_ENTRY('5', DIGIT, DIGIT, IDENT_ANYWHERE, NUMBER_START),  
  CHAR_ENTRY('6', DIGIT, DIGIT, IDENT_ANYWHERE, NUMBER_START),  
  CHAR_ENTRY('7', DIGIT, DIGIT, IDENT_ANYWHERE, NUMBER_START),  
  CHAR_ENTRY('8', DIGIT, DIGIT, IDENT_ANYWHERE, NUMBER_START),  
  CHAR_ENTRY('9', DIGIT, DIGIT, IDENT_ANYWHERE, NUMBER_START),  
  CHAR_ENTRY('<', OTHER, OPCHAR, OPCHAR_INFIX, ADDRESS_OP, TERMINATOR),  
  CHAR_ENTRY('>', OTHER, OPCHAR, OPCHAR_INFIX, TERMINATOR),  
  CHAR_ENTRY('=', OTHER, OPCHAR, OPCHAR_INFIX, TERMINATOR),  
  CHAR_ENTRY('.', OTHER, IDENT_ANYWHERE),  
  CHAR_ENTRY('$', OTHER, IDENT_ANYWHERE));
```

Define the COBOL Operator Table for operators whose names are identifiers.

```
OPERATOR TABLE(COBOL IDENT_OPTBL,  
  OPERATOR_ENTRY('OF', DOT, INFIX, 0, 0, PRIMARY),  
  OPERATOR_ENTRY('IN', DOT, INFIX, 0, 0, PRIMARY),  
  OPERATOR_ENTRY('NOT', NOT, PREFIX, 200, 25),  
  OPERATOR_ENTRY('AND', AND, INFIX, 20, 20),  
  OPERATOR_ENTRY('OR', OR, INFIX, 10, 10),  
  OPERATOR_ENTRY('NOT', INFIX_NOT, INFIX, 30, 30, LEXICAL));
```

BIND

```
COBOL NOT_EQUAL_TOKEN =  
  OPERATOR_ENTRY('NOT =', NOT_EQUAL, INFIX, 30, 30),  
COBOL NOT_GTR_TOKEN =  
  OPERATOR_ENTRY('NOT >', LSS_EQUAL, INFIX, 30, 30),  
COBOL NOT_LSS_TOKEN =  
  OPERATOR_ENTRY('NOT <', GTR_EQUAL, INFIX, 30, 30);
```

Define the COBOL Operator Table for operators whose names are composed of operator characters such as '+', '-', or '*'. This table includes operators which are part of DEBUG Primary Symbols (such as '\').

```
OPERATOR TABLE(COBOL OPCHAR_OPTBL,  
  OPERATOR_ENTRY('\', GLOBAL_SLASH, PREFIX, 0, 0, PRIMARY),  
  OPERATOR_ENTRY('\', BACKSLASH, INFIX, 0, 0, PRIMARY),  
  OPERATOR_ENTRY('(', SUBSCRIPT, POSTFIX, 0, 0, PRIMARY),
```

```
.. 2262 P 2386 1 OPERATOR_ENTRY('>', PREFIX_GTR, PREFIX, 200, 30, LEXICAL),
.. 2263 P 2387 1 OPERATOR_ENTRY('<', PREFIX_LSS, PREFIX, 200, 30, LEXICAL),
.. 2264 P 2388 1 OPERATOR_ENTRY('=', PREFIX_EQ, PREFIX, 200, 30, LEXICAL),
.. 2265 P 2389 1 OPERATOR_ENTRY('(', OPENPAREN, PREFIX, 200, 5, LEXICAL),
.. 2266 P 2390 1 OPERATOR_ENTRY(')', CLOSEPAREN, POSTFIX, 6, 200, LEXICAL),
.. 2267 P 2391 1 ! OPERATOR_ENTRY('^', POWER_OF, INFIX, 70, 70),
.. 2268 P 2392 1 ! OPERATOR_ENTRY('*', MULTIPLY, INFIX, 60, 60),
.. 2269 P 2393 1 ! OPERATOR_ENTRY('/', DIVIDE, INFIX, 60, 60),
.. 2270 P 2394 1 ! OPERATOR_ENTRY('+', UNARY_PLUS, PREFIX, 200, 50),
.. 2271 P 2395 1 ! OPERATOR_ENTRY('-', UNARY_MINUS, PREFIX, 200, 50),
.. 2272 P 2396 1 ! OPERATOR_ENTRY('+', ADD, INFIX, 40, 40),
.. 2273 P 2397 1 ! OPERATOR_ENTRY('-', SUBTRACT, INFIX, 40, 40),
.. 2274 P 2398 1 ! OPERATOR_ENTRY('>', GTR_THAN, INFIX, 30, 30),
.. 2275 P 2399 1 ! OPERATOR_ENTRY('<', LSS_THAN, INFIX, 30, 30),
.. 2276 2400 1 OPERATOR_ENTRY('=', EQUAL, INFIX, 30, 30);
.. 2277 2401 1
.. 2278 2402 1
.. 2279 2403 1 ! Define the COBOL Terminator Lexical Token Table for subscript expressions.
.. 2280 2404 1
.. 2281 P 2405 1 TERMINATOR_TABLE(COBOL_SUBSCR_TERM_TBL,
.. 2282 P 2406 1 TERMINATOR_ENTRY(')', TERM_CLOSE),
.. 2283 P 2407 1 TERMINATOR_ENTRY(':', TERM_COLON, MUST_BE_SINGLE),
.. 2284 2408 1 TERMINATOR_ENTRY(',', TERM_COMMA));
.. 2285 2409 1
.. 2286 2410 1
.. 2287 2411 1 ! Define the COBOL Predefined Identifier Table.
.. 2288 2412 1
.. 2289 2413 1 PRID_TABLE(COBOL_PRID_TABLE);
.. 2290 2414 1
.. 2291 2415 1
.. 2292 2416 1 ! Define the COBOL Built-in Function Table.
.. 2293 2417 1
.. 2294 2418 1 BUILT_IN_FUNCTION_TABLE(COBOL_FUNCTION_TABLE);
.. 2295 2419 1
.. 2296 2420 1
.. 2297 2421 1 ! Define the COBOL Number Scanner State Table. This table defines the states
.. 2298 2422 1 ! of a Finite-State Machine which picks up all valid numeric constants in the
.. 2299 2423 1 ! language. Each Transition Entry is of the form:
.. 2300 2424 1
.. 2301 2425 1 ! NUMBER_TRANSITION(character-class, action-index, next-state)
.. 2302 2426 1
.. 2303 2427 1 ! where the character-class and action-index names are automatically prefixed
.. 2304 2428 1 ! by 'NUMST$K_CLASS_' or 'NUMST$K_ACT_' by the NUMBER_TRANSITION macro.
.. 2305 2429 1
.. 2306 P 2430 1 NUMBER_STATE_TABLE(COBOL_NUMBER_TABLE,
.. 2307 P 2431 1
.. 2308 P 2432 1 NUMBER_STATE(START_STATE,
.. 2309 P 2433 1 NUMBER_TRANSITION(DIGIT, GO_PAST_PACK, ACCUM_INT),
.. 2310 P 2434 1 NUMBER_TRANSITION(DOT, DO_NOTHING, LEADING_DOT),
.. 2311 P 2435 1 NUMBER_TRANSITION(OTHER, NOT_NUMBER, END_STATE)),
.. 2312 P 2436 1
.. 2313 P 2437 1 NUMBER_STATE(LEADING_DOT,
.. 2314 P 2438 1 NUMBER_TRANSITION(DIGIT, GO_PAST_PACK_FRAC, ACCUM_FRAC),
.. 2315 P 2439 1 NUMBER_TRANSITION(OTHER, NOT_NUMBER, END_STATE)),
.. 2316 P 2440 1
.. 2317 P 2441 1 NUMBER_STATE(ACCUM_INT,
.. 2318 P 2442 1 NUMBER_TRANSITION(DIGIT, GO_PAST_PACK, ACCUM_INT),
```



```
2319 P 2443 1 NUMBER_TRANSITION(DOT, DO NOTHING, ACCUM_FRAC),
2320 P 2444 1 NUMBER_TRANSITION(HEXDIGIT, COB_CKHEX, ACCUM_HEX),
2321 P 2445 1 NUMBER_TRANSITION(B, COB_CKHEX, ACCUM_HEX),
2322 P 2446 1 NUMBER_TRANSITION(D, COB_CKHEX, ACCUM_HEX),
2323 P 2447 1 NUMBER_TRANSITION(E, COB_CKHEX, ACCUM_HEX),
2324 P 2448 1 NUMBER_TRANSITION(OTHER, COB_CKNUM, END_STATE)),
2325 P 2449 1
2326 P 2450 1 NUMBER_STATE(ACCUM_HEX,
2327 P 2451 1 NUMBER_TRANSITION(DIGIT, DO NOTHING, ACCUM_HEX),
2328 P 2452 1 NUMBER_TRANSITION(HEXDIGIT, DO NOTHING, ACCUM_HEX),
2329 P 2453 1 NUMBER_TRANSITION(B, DO NOTHING, ACCUM_HEX),
2330 P 2454 1 NUMBER_TRANSITION(D, DO NOTHING, ACCUM_HEX),
2331 P 2455 1 NUMBER_TRANSITION(E, DO NOTHING, ACCUM_HEX),
2332 P 2456 1 NUMBER_TRANSITION(OTHER, COB_CKNUM, END_STATE)),
2333 P 2457 1
2334 P 2458 1 NUMBER_STATE(ACCUM_FRAC,
2335 P 2459 1 NUMBER_TRANSITION(DIGIT, GO_PAST_PACK_FRAC, ACCUM_FRAC),
2336 P 2460 1 NUMBER_TRANSITION(DOT, BACKUP_PTRS, END_STATE),
2337 P 2461 1 NUMBER_TRANSITION(E, MARK_E_EXP, GET_EXPONENT),
2338 P 2462 1 NUMBER_TRANSITION(D, MARK_D_EXP, GET_EXPONENT),
2339 P 2463 1 NUMBER_TRANSITION(G, MARK_G_EXP, GET_EXPONENT),
2340 P 2464 1 NUMBER_TRANSITION(Q, MARK_Q_EXP, GET_EXPONENT),
2341 P 2465 1 NUMBER_TRANSITION(OTHER, GOT_PACK_NUMBER, END_STATE)),
2342 P 2466 1
2343 P 2467 1 NUMBER_STATE(GET_EXPONENT,
2344 P 2468 1 NUMBER_TRANSITION(DIGIT, DO NOTHING, ACCUM_EXP),
2345 P 2469 1 NUMBER_TRANSITION(PLUS, DO NOTHING, GET_EXP_SIGN),
2346 P 2470 1 NUMBER_TRANSITION(MINUS, DO NOTHING, GET_EXP_SIGN),
2347 P 2471 1 NUMBER_TRANSITION(OTHER, BACKUP_PTRS, END_STATE)),
2348 P 2472 1
2349 P 2473 1 NUMBER_STATE(GET_EXP_SIGN,
2350 P 2474 1 NUMBER_TRANSITION(DIGIT, DO NOTHING, ACCUM_EXP),
2351 P 2475 1 NUMBER_TRANSITION(OTHER, GOT_NUMBER, END_STATE)),
2352 P 2476 1
2353 P 2477 1 NUMBER_STATE(ACCUM_EXP,
2354 P 2478 1 NUMBER_TRANSITION(DIGIT, DO NOTHING, ACCUM_EXP),
2355 P 2479 1 NUMBER_TRANSITION(OTHER, GOT_NUMBER, END_STATE)),
2356 P 2480 1
2357 P 2481 1 NUMBER_STATE(END_STATE,
2358 2482 1 NUMBER_TRANSITION(OTHER, GIVE_ERROR, END_STATE)));
2359 2483 1
2360 2484 1
2361 2485 1 ! Define the Primary Parser State Table for language COBOL. Each transition
2362 2486 1 Entry in the state table has this format:
2363 2487 1
2364 2488 1 PRIMARY_TRANSITION(operator-code, action, next-state)
2365 2489 1
2366 2490 1 where the first parameter is the operator code which causes the transition
2367 2491 1 to be taken, the second parameter is the action routine CASE index for the
2368 2492 1 transition, and the third parameter is the next state in the finite-State
2369 2493 1 Machine.
2370 2494 1
2371 P 2495 1 PRIMARY_STATE_TABLE(COBOL_PRIMARY_TABLE,
2372 P 2496 1
2373 P 2497 1 PRIMARY_STATE(START_STATE,
2374 P 2498 1 PRIMARY_TRANSITION(GLOBAL_SLASH, START_GBL, GET_GLOBAL),
2375 P 2499 1 PRIMARY_TRANSITION(BACKSLASH, START_SLASH, GOT_BACKSLASH),
```

```
2376 P 2500 1 PRIMARY_TRANSITION(INVOCNUM, SLASH INVOCNUM, GOT_BACKSLASH),
2377 P 2501 1 PRIMARY_TRANSITION(DOT, START_DOT COB, GOT_DOT),
2378 P 2502 1 PRIMARY_TRANSITION(SUBSCRIPT, START SUBSCR PLI, GOT SUBSCRIPT),
2379 P 2503 1 PRIMARY_TRANSITION(PRIMARY_TERM, START_TERM, END_STATE)),
2380 P 2504 1
2381 P 2505 1 PRIMARY_STATE(GET GLOBAL,
2382 P 2506 1 PRIMARY_TRANSITION(PRIMARY_TERM, GBL_TERM, END_STATE)),
2383 P 2507 1
2384 P 2508 1 PRIMARY_STATE(GOT BACKSLASH,
2385 P 2509 1 PRIMARY_TRANSITION(BACKSLASH, SLASH SLASH, GOT BACKSLASH),
2386 P 2510 1 PRIMARY_TRANSITION(INVOCNUM, SLASH INVOCNUM, GOT BACKSLASH),
2387 P 2511 1 PRIMARY_TRANSITION(SUBSCRIPT, SLASH SUBSCR PLI, GOT SUBSCRIPT),
2388 P 2512 1 PRIMARY_TRANSITION(PRIMARY_TERM, SLASH_TERM, END_STATE)),
2389 P 2513 1
2390 P 2514 1 PRIMARY_STATE(GOT DOT,
2391 P 2515 1 PRIMARY_TRANSITION(BACKSLASH, DOT SLASH COB, GOT_BACKSLASH),
2392 P 2516 1 PRIMARY_TRANSITION(DOT, DOT DOT COB, GOT_DOT),
2393 P 2517 1 PRIMARY_TRANSITION(SUBSCRIPT, DOT SUBSCR COB, GOT SUBSCRIPT),
2394 P 2518 1 PRIMARY_TRANSITION(PRIMARY_TERM, DOT_TERM_COB, END_STATE)),
2395 P 2519 1
2396 P 2520 1 PRIMARY_STATE(GOT SUBSCRIPT,
2397 P 2521 1 PRIMARY_TRANSITION(SUBSCRIPT, SUBSCR SUBSCR PLI, GOT SUBSCRIPT2),
2398 P 2522 1 PRIMARY_TRANSITION(PRIMARY_TERM, SUBSCR_TERM_PLI, END_STATE)),
2399 P 2523 1
2400 P 2524 1 PRIMARY_STATE(GOT SUBSCRIPT2,
2401 P 2525 1 PRIMARY_TRANSITION(PRIMARY_TERM, SUBSCR_TERM_PLI, END_STATE)),
2402 P 2526 1
2403 P 2527 1 PRIMARY_STATE(END_STATE));
2404 P 2528 1
2405 P 2529 1
2406 P 2530 1 ! Define the table of pointers to the parse tables for COBOL.
2407 P 2531 1 !
2408 P 2532 1 LANGUAGE_TABLES(LANGUAGE = COBOL,
2409 P 2533 1 CHARTBL = COBOL CHARTBL,
2410 P 2534 1 IDENT OPTBL = COBOL IDENT OPTBL,
2411 P 2535 1 OPCHAR OPTBL = COBOL OPCHAR OPTBL,
2412 P 2536 1 NUMBER_TABLE = COBOL NUMBER_TABLE,
2413 P 2537 1 PRIMARY_TABLE = COBOL PRIMARY_TABLE,
2414 P 2538 1 SUBSCR_TERMS = COBOL SUBSCR_TERM_TBL,
2415 P 2539 1 PRIDTBL = COBOL PRID_TABLE,
2416 P 2540 1 BIF_TABLE = COBOL FUNCTION_TABLE,
2417 P 2541 1 MULTIPLE SUBSCR = FALSE,
2418 P 2542 1 COMPONENTS_IN_PATHNAME = TRUE);
```

F O R T R A N P A R S E T A B L E S

This section includes all the Lexical Scanner and Parser tables needed to scan and parse FORTRAN expressions.

Define the FORTRAN Character Table. What is listed here is actually a list of exceptions to the Character Table for language UNKNOWN. (Language UNKNOWN lists the "average" use of each character in the character set.)

P 2556 1 CHAR_EXCEPTION_TABLE(FORTRAN_CHARTBL,
P 2557 1 CHAR_ENTRY(' ', DOT, NUMBER_START, SPECIAL_SYMBOL),
2558 1 CHAR_ENTRY('/', OTHER, OPCHAR, OPCHAR_INFIX));

Define the FORTRAN Operator Table for operators whose names are identifiers. This table is empty since FORTRAN has no such operators, but the Lexical Scanner requires that such a table exist anyway.

OPERATOR_TABLE(FORTRAN_IDENT_OPTBL);

Define the FORTRAN Operator Table for operators whose names are composed of operator characters such as "+", "-", or "*". This table includes those operators which are part of DEBUG Primary Symbols (such as "\").

P 2572 1 OPERATOR_TABLE(FORTRAN_OPCHAR_OPTBL,
P 2573 1 OPERATOR_ENTRY('\', GLOBAL_SLASH, PREFIX, 0, 0, PRIMARY),
P 2574 1 OPERATOR_ENTRY('\'', BACKSLASH, INFIX, 0, 0, PRIMARY),
P 2575 1 OPERATOR_ENTRY('(', SUBSCRIPT, POSTFIX, 0, 0, PRIMARY),
P 2576 1 OPERATOR_ENTRY('.', DOT, INFIX, 0, 0, PRIMARY),
P 2577 1 OPERATOR_ENTRY('+', ADD, INFIX, 60, 60),
P 2578 1 OPERATOR_ENTRY('-', SUBTRACT, INFIX, 60, 60),
P 2579 1 OPERATOR_ENTRY('+', UNARY_PLUS, PREFIX, 200, 70),
P 2580 1 OPERATOR_ENTRY('-', UNARY_MINUS, PREFIX, 200, 70),
P 2581 1 OPERATOR_ENTRY('*', MULTIPLY, INFIX, 80, 80),
P 2582 1 OPERATOR_ENTRY('/', DIVIDE, INFIX, 80, 80),
P 2583 1 OPERATOR_ENTRY('**', POWER_OF, INFIX, 92, 90),
P 2584 1 OPERATOR_ENTRY('//', CONCATENATE, INFIX, 60, 60),
P 2585 1 OPERATOR_ENTRY('(', OPENPAREN, PREFIX, 200, 5, LEXICAL),
P 2586 1 OPERATOR_ENTRY(')', CLOSEPAREN, POSTFIX, 6, 200, LEXICAL));

BIND

FORTRAN_INDIRECT_TOKEN = OPERATOR_ENTRY('.', INDIRECT, PREFIX, 200, 40),
FORTRAN_DOT_TOKEN = OPERATOR_ENTRY('.', DOT, INFIX, 0, 0, PRIMARY);

Define an Operator Table for FORTRAN operators of the form .XX. or .XXX. This includes all the FORTRAN comparison and boolean operators.

P 2597 1 OPERATOR_TABLE(FORTRAN_SPECIAL_OPTBL,
P 2598 1 OPERATOR_ENTRY('EQ.', EQUAL, INFIX, 50, 50),
P 2599 1 OPERATOR_ENTRY('NE.', NOT_EQUAL, INFIX, 50, 50);


```
2477 P 2600 1 OPERATOR_ENTRY('GT.', GTR_THAN, INFIX, 50, 50),
2478 P 2601 1 OPERATOR_ENTRY('GE.', GTR_EQUAL, INFIX, 50, 50),
2479 P 2602 1 OPERATOR_ENTRY('LT.', LSS_THAN, INFIX, 50, 50),
2480 P 2603 1 OPERATOR_ENTRY('LE.', LSS_EQUAL, INFIX, 50, 50),
2481 P 2604 1 OPERATOR_ENTRY('NOT.', NOT, PREFIX, 200, 40),
2482 P 2605 1 OPERATOR_ENTRY('AND.', AND, INFIX, 30, 30),
2483 P 2606 1 OPERATOR_ENTRY('OR.', OR, INFIX, 20, 20),
2484 P 2607 1 OPERATOR_ENTRY('XOR.', XOR, INFIX, 10, 10),
2485 P 2608 1 OPERATOR_ENTRY('EQV.', EQV, INFIX, 10, 10),
2486 P 2609 1 OPERATOR_ENTRY('NEQV.', XOR, INFIX, 10, 10));
2487 2610 1
2488 2611 1
2489 2612 1 ! Define the FORTRAN Terminator Lexical Token Table for subscript expressions.
2490 2613 1 ! In FORTRAN, a subscript expression can be terminated by ')' (end of sub-
2491 2614 1 ! scripts), by ',' (more subscripts to follow), or by ':' (string subscript
2492 2615 1 ! upper bound to follow).
2493 2616 1
2494 P 2617 1 TERMINATOR_TABLE(FORTRAN_SUBSCR_TERM_TBL,
2495 P 2618 1 TERMINATOR_ENTRY(')', TERM_CLOSE, BALANCED_PARENS),
2496 P 2619 1 TERMINATOR_ENTRY(',', TERM_COMMA),
2497 2620 1 TERMINATOR_ENTRY(':', TERM_COLON));
2498 2621 1
2499 2622 1
2500 2623 1 ! Define the FORTRAN Predefined Identifier Table.
2501 2624 1
2502 P 2625 1 PRID_TABLE(FORTRAN_PRID_TABLE,
2503 P 2626 1 PRID_ENTRY('TRUE.', ATOMIC, L, 1),
2504 2627 1 PRID_ENTRY('FALSE.', ATOMIC, L, 0));
2505 2628 1
2506 2629 1
2507 2630 1 ! Define the FORTRAN Built-in Function Table.
2508 2631 1
2509 2632 1 BUILT_IN_FUNCTION_TABLE(FORTRAN_FUNCTION_TABLE);
2510 2633 1
2511 2634 1
2512 2635 1 ! Define the FORTRAN Number Scanner State Table. This table defines the states
2513 2636 1 ! of a Finite-State Machine which picks up all valid numeric constants in the
2514 2637 1 ! language.
2515 2638 1
2516 2639 1 ! The FORTRAN number table is the same as the number table for language UNKNOWN.
2517 2640 1
2518 2641 1 BIND
2519 2642 1 FORTRAN_NUMBER_TABLE = UNKNOWN_NUMBER_TABLE;
2520 2643 1
2521 2644 1
2522 2645 1 ! Define the Primary Parser State Table for language FORTRAN. Each Transition
2523 2646 1 ! Entry in the state table has this format:
2524 2647 1
2525 2648 1 PRIMARY_TRANSITION(operator-code, action, next-state)
2526 2649 1
2527 2650 1 ! where the first parameter is the operator code which causes the transition
2528 2651 1 ! to be taken, the second parameter is the action routine CASE index for the
2529 2652 1 ! transition, and the third parameter is the next state in the Finite-State
2530 2653 1 ! Machine.
2531 2654 1
2532 P 2655 1 PRIMARY_STATE_TABLE(FORTRAN_PRIMARY_TABLE,
2533 P 2656 1
```

```
2534 P 2657 1 PRIMARY_STATE(START_STATE,  
2535 P 2658 1 PRIMARY_TRANSITION(GLOBAL_SLASH, START_GBL, GET_GLOBAL),  
2536 P 2659 1 PRIMARY_TRANSITION(BACKSLASH, START_SLASH, GOT_BACKSLASH),  
2537 P 2660 1 PRIMARY_TRANSITION(INVOCNUM, SLASH_INVOCNUM, GOT_BACKSLASH),  
2538 P 2661 1 PRIMARY_TRANSITION(DOT, START_DOT, GOT_DOT),  
2539 P 2662 1 PRIMARY_TRANSITION(SUBSCRIPT, START_SUBSCR, GOT_SUBSCRIPT),  
2540 P 2663 1 PRIMARY_TRANSITION(PRIMARY_TERM, START_TERM, END_STATE)),  
2541 P 2664 1  
2542 P 2665 1 PRIMARY_STATE(GET_GLOBAL,  
2543 P 2666 1 PRIMARY_TRANSITION(PRIMARY_TERM, GBL_TERM, END_STATE)),  
2544 P 2667 1  
2545 P 2668 1 PRIMARY_STATE(GOT_BACKSLASH,  
2546 P 2669 1 PRIMARY_TRANSITION(BACKSLASH, SLASH_SLASH, GOT_BACKSLASH),  
2547 P 2670 1 PRIMARY_TRANSITION(INVOCNUM, SLASH_INVOCNUM, GOT_BACKSLASH),  
2548 P 2671 1 PRIMARY_TRANSITION(SUBSCRIPT, SLASH_SUBSCR, GOT_SUBSCRIPT),  
2549 P 2672 1 PRIMARY_TRANSITION(PRIMARY_TERM, SLASH_TERM, END_STATE)),  
2550 P 2673 1  
2551 P 2674 1 PRIMARY_STATE(GOT_DOT,  
2552 P 2675 1 PRIMARY_TRANSITION(DOT, DOT_DOT, GOT_DOT),  
2553 P 2676 1 PRIMARY_TRANSITION(SUBSCRIPT, DOT_SUBSCR, GOT_SUBSCRIPT),  
2554 P 2677 1 PRIMARY_TRANSITION(PRIMARY_TERM, DOT_TERM, END_STATE)),  
2555 P 2678 1  
2556 P 2679 1 PRIMARY_STATE(GOT_SUBSCRIPT,  
2557 P 2680 1 PRIMARY_TRANSITION(DOT, SUBSCR_DOT, GOT_DOT),  
2558 P 2681 1 PRIMARY_TRANSITION(SUBSCRIPT, SUBSCR_SUBSCR, GOT_SUBSCRIPT),  
2559 P 2682 1 PRIMARY_TRANSITION(PRIMARY_TERM, SUBSCR_TERM, END_STATE)),  
2560 P 2683 1  
2561 2684 1 PRIMARY_STATE(END_STATE));  
2562 2685 1  
2563 2686 1  
2564 2687 1 ! Define the table of pointers to the parse tables for FORTRAN.  
2565 2688 1 !  
2566 P 2689 1 LANGUAGE_TABLES(LANGUAGE = FORTRAN,  
2567 P 2690 1 CHARTBL = FORTRAN_CHARTBL,  
2568 P 2691 1 IDENT_OPTBL = FORTRAN_IDENT_OPTBL,  
2569 P 2692 1 OPCHAR_OPTBL = FORTRAN_OPCHAR_OPTBL,  
2570 P 2693 1 NUMBER_TABLE = FORTRAN_NUMBER_TABLE,  
2571 P 2694 1 PRIMARY_TABLE = FORTRAN_PRIMARY_TABLE,  
2572 P 2695 1 SUBSCR_TERMS = FORTRAN_SUBSCR_TERM_TBL,  
2573 P 2696 1 PRIDTBL = FORTRAN_PRID_TABLE,  
2574 2697 1 BIF_TABLE = FORTRAN_FUNCTION_TABLE);
```

MACRO PARSE TABLES

This section includes all the Lexical Scanner and Parser tables needed to scan and parse the MACRO language.

Define the MACRO Character Table. What is listed here is actually a list of exceptions to the Character Table for Language UNKNOWN.

```
CHAR_EXCEPTION TABLE(MACRO CHARTBL,  
  CHAR_ENTRY(' ', OTHER, IDENT_ANYWHERE),  
  CHAR_ENTRY('$', OTHER, IDENT_ANYWHERE),  
  CHAR_ENTRY('.', DOT, NUMBER_START, OPCHAR, ADDRESS_OP, SPECIAL_SYMBOL,  
    IDENT_MIDDLE, IDENT_END),  
  CHAR_ENTRY('<', OTHER, OPCHAR, ADDRESS_OP),  
  CHAR_ENTRY('>', OTHER, TERMINATOR));
```

Define the MACRO Operator Table for operators whose names are identifiers.

```
OPERATOR TABLE(MACRO IDENT OPTBL,  
  OPERATOR_ENTRY('EQ', EQUAL, INFIX, 50, 50),  
  OPERATOR_ENTRY('EQU', EQUAL, INFIX, 50, 50),  
  OPERATOR_ENTRY('NEQ', NOT_EQUAL, INFIX, 50, 50),  
  OPERATOR_ENTRY('NEQU', NOT_EQUAL, INFIX, 50, 50),  
  OPERATOR_ENTRY('GTR', GTR_THAN, INFIX, 50, 50),  
  OPERATOR_ENTRY('GTRU', GTR_THAN_U, INFIX, 50, 50),  
  OPERATOR_ENTRY('GEQ', GTR_EQUAL, INFIX, 50, 50),  
  OPERATOR_ENTRY('GEQU', GTR_EQUAL_U, INFIX, 50, 50),  
  OPERATOR_ENTRY('LSS', LSS_THAN, INFIX, 50, 50),  
  OPERATOR_ENTRY('LSSU', LSS_THAN_U, INFIX, 50, 50),  
  OPERATOR_ENTRY('LEQ', LSS_EQUAL, INFIX, 50, 50),  
  OPERATOR_ENTRY('LEQU', LSS_EQUAL_U, INFIX, 50, 50),  
  OPERATOR_ENTRY('NOT', BIT_NOT, PREFIX, 200, 40),  
  OPERATOR_ENTRY('AND', BIT_AND, INFIX, 30, 30),  
  OPERATOR_ENTRY('OR', BIT_OR, INFIX, 20, 20),  
  OPERATOR_ENTRY('EQV', BIT_EQV, INFIX, 10, 10),  
  OPERATOR_ENTRY('XOR', BIT_XOR, INFIX, 10, 10),  
  OPERATOR_ENTRY('MOD', REMAINDER, INFIX, 70, 70));
```

Define the MACRO Operator Table for operators whose names are composed of operator characters such as '+', '-', or '*'. This table includes operators which are part of DEBUG Primary Symbols (such as '\').

```
OPERATOR TABLE(MACRO OPCHAR OPTBL,  
  OPERATOR_ENTRY('\ ', GLOBAL_SLASH, PREFIX, 0, 0, PRIMARY),  
  OPERATOR_ENTRY('\ ', BACKSLASH, INFIX, 0, 0, PRIMARY),  
  
  OPERATOR_ENTRY('(', OPENPAREN, PREFIX, 200, 5, LEXICAL),  
  OPERATOR_ENTRY(')', CLOSEPAREN, POSTFIX, 6, 200, LEXICAL),  
  OPERATOR_ENTRY('<', BITSELECT, POSTFIX, 110, 200, LEXICAL),  
  OPERATOR_ENTRY('@', INDIRECT, PREFIX, 200, 100),  
  OPERATOR_ENTRY('@', INDIRECT, PREFIX, 200, 100));
```



```
2633 P 2755 1 OPERATOR_ENTRY('+', UNARY_PLUS, PREFIX, 200, 90),
2634 P 2756 1 OPERATOR_ENTRY('-', UNARY_MINUS, PREFIX, 200, 90),
2635 P 2757 1 OPERATOR_ENTRY('@', LEFT_SHIFT, INFIX, 80, 80),
2636 P 2758 1 OPERATOR_ENTRY('*', MULTIPLY, INFIX, 70, 70),
2637 P 2759 1 OPERATOR_ENTRY('/', DIVIDE, INFIX, 70, 70),
2638 P 2760 1 OPERATOR_ENTRY('+', ADD, INFIX, 60, 60),
2639 P 2761 1 OPERATOR_ENTRY('-', SUBTRACT, INFIX, 60, 60),
2640 2762 1 );
2641 2763 1
2642 2764 1
2643 2765 1 ! Define the MACRO Terminator Lexical Token Table for subscript expressions.
2644 2766 1 ! Since MACRO does not have subscripting, this table is empty.
2645 2767 1
2646 2768 1 TERMINATOR_TABLE(MACRO_SUBSCR_TERM_TBL);
2647 2769 1
2648 2770 1
2649 2771 1 ! Define the MACRO Predefined Identifier Table.
2650 2772 1
2651 2773 1 PRID_TABLE(MACRO_PRID_TABLE);
2652 2774 1
2653 2775 1
2654 2776 1 ! Define the MACRO Built-in Function Table.
2655 2777 1
2656 2778 1 BUILT_IN_FUNCTION_TABLE(MACRO_FUNCTION_TABLE);
2657 2779 1
2658 2780 1
2659 2781 1 ! Define the MACRO Number Scanner State Table. This table defines the states
2660 2782 1 ! of a Finite-State Machine which picks up all valid numeric constants in the
2661 2783 1 ! language.
2662 2784 1
2663 2785 1 ! The MACRO number table is the same as the number table for language UNKNOWN.
2664 2786 1
2665 2787 1 BIND
2666 2788 1 MACRO_NUMBER_TABLE = UNKNOWN_NUMBER_TABLE;
2667 2789 1
2668 2790 1
2669 2791 1 ! Define the Primary Parser State Table for language MACRO. Each transition
2670 2792 1 ! entry in the state table has this format:
2671 2793 1
2672 2794 1 PRIMARY_TRANSITION(operator-code, action, next-state)
2673 2795 1
2674 2796 1 ! where the first parameter is the operator code which causes the transition
2675 2797 1 ! to be taken, the second parameter is the action routine CASE index for the
2676 2798 1 ! transition, and the third parameter is the next state in the Finite-State
2677 2799 1 ! Machine.
2678 2800 1
2679 P 2801 1 PRIMARY_STATE_TABLE(MACRO_PRIMARY_TABLE,
2680 P 2802 1
2681 P 2803 1 PRIMARY_STATE(START_STATE,
2682 P 2804 1 PRIMARY_TRANSITION(GLOBAL_SLASH, START_GBL, GET_GLOBAL),
2683 P 2805 1 PRIMARY_TRANSITION(BACKSLASH, START_SLASH, GOT_BACKSLASH),
2684 P 2806 1 PRIMARY_TRANSITION(INVOCNUM, SLASH_INVOCNUM, GOT_BACKSLASH),
2685 P 2807 1 PRIMARY_TRANSITION(PRIMARY_TERM, START_TERM, END_STATE)),
2686 P 2808 1
2687 P 2809 1 PRIMARY_STATE(GET_GLOBAL,
2688 P 2810 1 PRIMARY_TRANSITION(PRIMARY_TERM, GBL_TERM, END_STATE)),
2689 P 2811 1
```

```

: 2690      P 2812 1    PRIMARY STATE(GOT BACKSLASH,
: 2691      P 2813 1    PRIMARY_TRANSITION(BACKSLASH, SLASH SLASH, GOT BACKSLASH),
: 2692      P 2814 1    PRIMARY_TRANSITION(INVOCNUM, SLASH INVOCNUM, GOT BACKSLASH),
: 2693      P 2815 1    PRIMARY_TRANSITION(PRIMARY_TERM, SLASH_TERM, END_STATE)),
: 2694      P 2816 1
: 2695      P 2817 1    PRIMARY_STATE(END_STATE));
: 2696      P 2818 1
: 2697      P 2819 1
: 2698      P 2820 1    ! Define the table of pointers to the parse tables for MACRO.
: 2699      P 2821 1    !
: 2700      P 2822 1    LANGUAGE_TABLES(LANGUAGE = MACRO,
: 2701      P 2823 1    CHARTBL = MACRO CHARTBL,
: 2702      P 2824 1    IDENT_OPTBL = MACRO IDENT_OPTBL,
: 2703      P 2825 1    OPCHAR_OPTBL = MACRO OPCHAR_OPTBL,
: 2704      P 2826 1    NUMBER_TABLE = MACRO NUMBER_TABLE,
: 2705      P 2827 1    PRIMARY_TABLE = MACRO PRIMARY_TABLE,
: 2706      P 2828 1    SUBSCR_TERMS = MACRO SUBSCR_TERM_TBL,
: 2707      P 2829 1    PRIDTBL = MACRO PRID_TABLE,
: 2708      P 2830 1    BIF_TABLE = MACRO_FUNCTION_TABLE);

```

P A S C A L P A R S E T A B L E S

This section includes all the Lexical Scanner and Parser tables needed to scan and parse the PASCAL language.

Define the PASCAL Character Table. What is listed here is actually a list of exceptions to the Character Table for Language UNKNOWN.

```
CHAR_EXCEPTION TABLE(PASCAL_CHARTBL,  
  CHAR_ENTRY('<', OTHER, OPCHAR, OPCHAR_INFIX, ADDRESS_OP),  
  CHAR_ENTRY('=', OTHER, OPCHAR, OPCHAR_INFIX, TERMINATOR),  
  CHAR_ENTRY('>', OTHER, OPCHAR, OPCHAR_INFIX, TERMINATOR),  
  CHAR_ENTRY('^', OTHER, OPCHAR, SPECIAL_SYMBOL),  
  CHAR_ENTRY('.', DOT, NUMBER_START, OPCHAR, ADDRESS_OP, SPECIAL_SYMBOL, TERMINATOR),  
  CHAR_ENTRY('[', OTHER, OPCHAR,  
  CHAR_ENTRY(']', OTHER, TERMINATOR));
```

! Define the PASCAL Operator Table for operators whose names are identifiers.

```
OPERATOR TABLE(PASCAL_IDENT_OPTBL,  
  OPERATOR_ENTRY('DIV', INT_DIVIDE, INFIX, 70, 70),  
  OPERATOR_ENTRY('MOD', MODULUS, INFIX, 70, 70),  
  OPERATOR_ENTRY('REM', REMAINDER, INFIX, 70, 70),  
  OPERATOR_ENTRY('AND', AND, INFIX, 70, 70),  
  OPERATOR_ENTRY('OR', OR, INFIX, 60, 60),  
  OPERATOR_ENTRY('NOT', NOT, PREFIX, 200, 90),  
  OPERATOR_ENTRY('IN', SET_MEMBER, INFIX, 50, 50));
```

! Define the PASCAL Operator Table for operators whose names are composed of operator characters such as "+", "-", or "*". This table includes operators which are part of DEBUG Primary Symbols (such as "\").

```
OPERATOR TABLE(PASCAL_OPCHAR_OPTBL,  
  OPERATOR_ENTRY('\', GLOBAL_SLASH, PREFIX, 0, 0, PRIMARY),  
  OPERATOR_ENTRY('\', BACKSLASH, INFIX, 0, 0, PRIMARY),  
  OPERATOR_ENTRY('[', SUBSCRIPT, POSTFIX, 0, 0, PRIMARY),  
  OPERATOR_ENTRY('.', DOT, INFIX, 0, 0, PRIMARY),  
  OPERATOR_ENTRY('^', PASCAL_DEREF, POSTFIX, 0, 0, PRIMARY),  
  OPERATOR_ENTRY('(', BIF_OP, POSTFIX, 0, 0, PRIMARY),  
  
  OPERATOR_ENTRY('(', OPENPAREN, PREFIX, 200, 5, LEXICAL),  
  OPERATOR_ENTRY(')', CLOSEPAREN, POSTFIX, 6, 200, LEXICAL),  
  OPERATOR_ENTRY('[', OPENSET, PREFIX, 200, 5, LEXICAL),  
  OPERATOR_ENTRY('+', UNARY_PLUS, PREFIX, 200, 60),  
  OPERATOR_ENTRY('-', UNARY_MINUS, PREFIX, 200, 60),  
  OPERATOR_ENTRY('*', POWER_OF, INFIX, 80, 80),  
  OPERATOR_ENTRY('*', MULTIPLY, INFIX, 70, 70),  
  OPERATOR_ENTRY('/', DIVIDE, INFIX, 70, 70),  
  OPERATOR_ENTRY('+', ADD, INFIX, 60, 60),  
  OPERATOR_ENTRY('-', SUBTRACT, INFIX, 60, 60),  
  OPERATOR_ENTRY('<', LSS_THAN, INFIX, 50, 50),
```



```
2767 P 2888 1 OPERATOR_ENTRY('<=', LSS_EQUAL, INFIX, 50, 50),
2768 P 2889 1 OPERATOR_ENTRY('>', GTR_THAN, INFIX, 50, 50),
2769 P 2890 1 OPERATOR_ENTRY('>=', GTR_EQUAL, INFIX, 50, 50),
2770 P 2891 1 OPERATOR_ENTRY('=', EQUAL, INFIX, 50, 50),
2771 2892 1 OPERATOR_ENTRY('<>', NOT_EQUAL, INFIX, 50, 50));
2772 2893 1
2773 2894 1
2774 2895 1 ! Define the PASCAL Terminator Lexical Token Table for subscript expressions.
2775 2896 1
2776 P 2897 1 TERMINATOR_TABLE(PASCAL_SUBSCR_TERM_TBL,
2777 P 2898 1 TERMINATOR_ENTRY(']', TERM_CLOSE),
2778 P 2899 1 TERMINATOR_ENTRY(':', TERM_COLON, MUST_BE_SINGLE),
2779 2900 1 TERMINATOR_ENTRY(',', TERM_COMMA));
2780 2901 1
2781 2902 1
2782 2903 1 ! Define the PASCAL Predefined Identifier Table.
2783 2904 1
2784 P 2905 1 PRID_TABLE(PASCAL_PRID_TABLE,
2785 P 2906 1 PRID_ENTRY('TRUE', ATOMIC, TF, 1),
2786 P 2907 1 PRID_ENTRY('FALSE', ATOMIC, TF, 0),
2787 2908 1 PRID_ENTRY('NIL', IPTR, Z, 0));
2788 2909 1
2789 2910 1
2790 2911 1 ! Define the PASCAL Built-in Function Table.
2791 2912 1
2792 2913 1 BUILT_IN_FUNCTION_TABLE(PASCAL_FUNCTION_TABLE);
2793 2914 1
2794 2915 1
2795 2916 1 ! Define the PASCAL Number Scanner State Table. This table defines the states
2796 2917 1 of a Finite-State Machine which picks up all valid numeric constants in the
2797 2918 1 language.
2798 2919 1
2799 2920 1 The PASCAL number table is the same as the number table for language UNKNOWN.
2800 2921 1
2801 2922 1 BIND
2802 2923 1 PASCAL_NUMBER_TABLE = UNKNOWN_NUMBER_TABLE;
2803 2924 1
2804 2925 1
2805 2926 1 ! Define the Primary Parser State Table for language PASCAL. Each transition
2806 2927 1 Entry in the state table has this format:
2807 2928 1
2808 2929 1 PRIMARY_TRANSITION(operator-code, action, next-state)
2809 2930 1
2810 2931 1 where the first parameter is the operator code which causes the transition
2811 2932 1 to be taken, the second parameter is the action routine CASE index for the
2812 2933 1 transition, and the third parameter is the next state in the finite-State
2813 2934 1 Machine.
2814 2935 1
2815 P 2936 1 PRIMARY_STATE_TABLE(PASCAL_PRIMARY_TABLE,
2816 P 2937 1
2817 P 2938 1 PRIMARY_STATE(START_STATE,
2818 P 2939 1 PRIMARY_TRANSITION(GLOBAL_SLASH, START_GBL, GET_GLOBAL),
2819 P 2940 1 PRIMARY_TRANSITION(BACKSLASH, START_SLASH, GOT_BACKSLASH),
2820 P 2941 1 PRIMARY_TRANSITION(INVOCNUM, SLASH_INVOCNUM, GOT_BACKSLASH),
2821 P 2942 1 PRIMARY_TRANSITION(DOT, START_DOT, GOT_DOT),
2822 P 2943 1 PRIMARY_TRANSITION(SUBSCRIPT, START_SUBSCR, GOT_SUBSCRIPT),
2823 P 2944 1 PRIMARY_TRANSITION(PASCAL_DEREF, START_DEREF, GOT_DEREF),
```

```
2824 P 2945 1 PRIMARY_TRANSITION(BIF OP, START_BIF CALL, END_STATE),
2825 P 2946 1 PRIMARY_TRANSITION(PRIMARY_TERM, START_TERM, END_STATE)),
2826 P 2947 1
2827 P 2948 1 PRIMARY_STATE(GET GLOBAL,
2828 P 2949 1 PRIMARY_TRANSITION(PRIMARY_TERM, GBL_TERM, END_STATE)),
2829 P 2950 1
2830 P 2951 1 PRIMARY_STATE(GOT BACKSLASH,
2831 P 2952 1 PRIMARY_TRANSITION(BACKSLASH, SLASH SLASH, GOT BACKSLASH),
2832 P 2953 1 PRIMARY_TRANSITION(INVOCNUM, SLASH_INVOCNUM, GOT BACKSLASH),
2833 P 2954 1 PRIMARY_TRANSITION(DOT, SLASH DOT, GOT DOT),
2834 P 2955 1 PRIMARY_TRANSITION(SUBSCRIPT, SLASH SUBSCR, GOT SUBSCRIPT),
2835 P 2956 1 PRIMARY_TRANSITION(PASCAL Deref, SLASH Deref, GOT Deref),
2836 P 2957 1 PRIMARY_TRANSITION(PRIMARY_TERM, SLASH_TERM, END_STATE)),
2837 P 2958 1
2838 P 2959 1 PRIMARY_STATE(GOT DOT,
2839 P 2960 1 PRIMARY_TRANSITION(DOT, DOT DOT, GOT DOT),
2840 P 2961 1 PRIMARY_TRANSITION(SUBSCRIPT, DOT SUBSCR, GOT SUBSCRIPT),
2841 P 2962 1 PRIMARY_TRANSITION(PASCAL Deref, DOT Deref, GOT Deref),
2842 P 2963 1 PRIMARY_TRANSITION(PRIMARY_TERM, DOT_TERM, END_STATE)),
2843 P 2964 1
2844 P 2965 1 PRIMARY_STATE(GOT SUBSCRIPT,
2845 P 2966 1 PRIMARY_TRANSITION(DOT, SUBSCR DOT, GOT DOT),
2846 P 2967 1 PRIMARY_TRANSITION(SUBSCRIPT, SUBSCR SUBSCR, GOT SUBSCRIPT),
2847 P 2968 1 PRIMARY_TRANSITION(PASCAL Deref, SUBSCR Deref, GOT Deref),
2848 P 2969 1 PRIMARY_TRANSITION(PRIMARY_TERM, SUBSCR_TERM, END_STATE)),
2849 P 2970 1
2850 P 2971 1 PRIMARY_STATE(GOT Deref,
2851 P 2972 1 PRIMARY_TRANSITION(DOT, Deref DOT, GOT DOT),
2852 P 2973 1 PRIMARY_TRANSITION(SUBSCRIPT, Deref SUBSCR, GOT SUBSCRIPT),
2853 P 2974 1 PRIMARY_TRANSITION(PASCAL Deref, Deref Deref, GOT Deref),
2854 P 2975 1 PRIMARY_TRANSITION(PRIMARY_TERM, Deref_TERM, END_STATE)),
2855 P 2976 1
2856 P 2977 1 PRIMARY_STATE(END_STATE));
2857 P 2978 1
2858 P 2979 1
2859 P 2980 1 ! Define the table of pointers to the parse tables for PASCAL.
2860 P 2981 1 !
2861 P 2982 1 LANGUAGE_TABLES(LANGUAGE = PASCAL,
2862 P 2983 1 CHARTBL = PASCAL_CHARTBL,
2863 P 2984 1 IDENT_OPTBL = PASCAL_IDENT_OPTBL,
2864 P 2985 1 OPCHAR_OPTBL = PASCAL_OPCHAR_OPTBL,
2865 P 2986 1 NUMBER_TABLE = PASCAL_NUMBER_TABLE,
2866 P 2987 1 PRIMARY_TABLE = PASCAL_PRIMARY_TABLE,
2867 P 2988 1 SUBSCR_TERMS = PASCAL_SUBSCR_TERM_TBL,
2868 P 2989 1 PRIDTBL = PASCAL_PRID_TABLE,
2869 P 2990 1 BIF_TABLE = PASCAL_FUNCTION_TABLE,
2870 P 2991 1 MULTIPLE_SUBSCR = TRUE);
```

PL / I PARSE TABLES

This section includes all the Lexical Scanner and Parser tables needed to scan and parse the PL/I language.

Define the PL/I Character Table. What is listed here is actually a list of exceptions to the Character Table for Language UNKNOWN.

CHAR_EXCEPTION TABLE(PLI CHARTBL,

CHAR_ENTRY('S', OTHER, IDENT_ANYWHERE),
CHAR_ENTRY(' ', OTHER, IDENT_ANYWHERE),
CHAR_ENTRY('%', OTHER, IDENT_MIDDLE),
CHAR_ENTRY('&', OTHER, OPCHAR),
CHAR_ENTRY('!', OTHER, OPCHAR, OPCHAR_INFIX),
CHAR_ENTRY(':', OTHER, OPCHAR, OPCHAR_INFIX),
CHAR_ENTRY('=', OTHER, OPCHAR, OPCHAR_INFIX, TERMINATOR),
CHAR_ENTRY('^', OTHER, OPCHAR, OPCHAR_INFIX, SPECIAL_SYMBOL),
CHAR_ENTRY('-', MINUS, OPCHAR, ADDRESS_OP, SPECIAL_CASE),
CHAR_ENTRY('<', OTHER, OPCHAR, OPCHAR_INFIX, ADDRESS_OP),
CHAR_ENTRY('>', OTHER, OPCHAR, OPCHAR_INFIX, TERMINATOR));

Define the PL/I Operator Table for operators whose names are identifiers.

OPERATOR_TABLE(PLI_IDENT_OPTBL);

Define the PL/I Operator Table for operators whose names are composed of operator characters such as '+', '-', or '*'. This table includes operators which are part of DEBUG Primary Symbols (such as '\').

OPERATOR_TABLE(PLI_OPCHAR_OPTBL,

OPERATOR_ENTRY('\', GLOBAL_SLASH, PREFIX, 0, 0, PRIMARY),
OPERATOR_ENTRY('\'', BACKSLASH, INFIX, 0, 0, PRIMARY),
OPERATOR_ENTRY('(', SUBSCRIPT, POSTFIX, 0, 0, PRIMARY),
OPERATOR_ENTRY('.', DOT, INFIX, 0, 0, PRIMARY),

OPERATOR_ENTRY('(', OPENPAREN, PREFIX, 200, 5, LEXICAL),
OPERATOR_ENTRY(')', CLOSEPAREN, POSTFIX, 6, 200, LEXICAL),
OPERATOR_ENTRY('+', UNARY_PLUS, PREFIX, 200, 70),
OPERATOR_ENTRY('-', UNARY_MINUS, PREFIX, 200, 70),
OPERATOR_ENTRY('^', BIT_NOT, PREFIX, 200, 30),
OPERATOR_ENTRY('*', POWER_OF, INFIX, 92, 90),
OPERATOR_ENTRY('*', MULTIPLY, INFIX, 80, 80),
OPERATOR_ENTRY('/', DIVIDE, INFIX, 80, 80),
OPERATOR_ENTRY('+', ADD, INFIX, 60, 60),
OPERATOR_ENTRY('-', SUBTRACT, INFIX, 60, 60),
OPERATOR_ENTRY('||', CONCATENATE, INFIX, 55, 55),
OPERATOR_ENTRY('>', GTR_THAN, INFIX, 50, 50),
OPERATOR_ENTRY('<', LSS_THAN, INFIX, 50, 50),
OPERATOR_ENTRY('>=', LSS_EQUAL, INFIX, 50, 50),
OPERATOR_ENTRY('<=', GTR_EQUAL, INFIX, 50, 50),
OPERATOR_ENTRY('=', EQUAL, INFIX, 50, 50).


```
.. 2929 P 3049 1 OPERATOR_ENTRY('!=', NOT_EQUAL, INFIX, 50, 50),
2930 P 3050 1 OPERATOR_ENTRY('<=', LSS_EQUAL, INFIX, 50, 50),
2931 P 3051 1 OPERATOR_ENTRY('>=', GTR_EQUAL, INFIX, 50, 50),
2932 P 3052 1 OPERATOR_ENTRY('&', BIT_AND, INFIX, 45, 45),
2933 3053 1 OPERATOR_ENTRY('|', BIT_OR, INFIX, 40, 40));
2934 3054 1
2935 3055 1
2936 3056 1 ! Define the Lexical Token Entry for the PL/I dereference operator '->'.
2937 3057 1 ! This token is lexically scanned separately.
2938 3058 1
2939 3059 1 BIND
2940 3060 1 PLI_ARROW_TOKEN =
2941 3061 1 OPERATOR_ENTRY('->', PLI_DEREF, INFIX, 0, 0, PRIMARY);
2942 3062 1
2943 3063 1
2944 3064 1 ! Define the PL/I Terminator Lexical Token Table for subscript expressions.
2945 3065 1
2946 P 3066 1 TERMINATOR_TABLE(PLI_SUBSCR_TERM_TBL,
2947 P 3067 1 TERMINATOR_ENTRY(')', TERM_CLOSE),
2948 P 3068 1 TERMINATOR_ENTRY(':', TERM_COLON, MUST_BE_SINGLE),
2949 3069 1 TERMINATOR_ENTRY(',', TERM_COMMA));
2950 3070 1
2951 3071 1
2952 3072 1 ! Define the PL/I Predefined Identifier Table.
2953 3073 1
2954 3074 1 PRID_TABLE(PLI_PRID_TABLE);
2955 3075 1
2956 3076 1
2957 3077 1 ! Define the PL/I Built-in Function Table.
2958 3078 1
2959 3079 1 BUILT_IN_FUNCTION_TABLE(PLI_FUNCTION_TABLE);
2960 3080 1
2961 3081 1
2962 3082 1 ! Define the PL/I Number Scanner State Table. This table defines the states
2963 3083 1 of a Finite-State Machine which picks up all valid numeric constants in the
2964 3084 1 language.
2965 3085 1
2966 3086 1 ! Define the language PL/I Number Scanner State Table. This is a finite-state
2967 3087 1 machine in which each transition is of the form:
2968 3088 1
2969 3089 1 NUMBER_TRANSITION(character-class, action-index, next-state)
2970 3090 1
2971 3091 1 where the character-class and action-index names are automatically prefixed
2972 3092 1 by 'NUMSTSK_CLASS_' or 'NUMSTSK_ACT_' by the NUMBER_TRANSITION macro.
2973 3093 1
2974 P 3094 1 NUMBER_STATE_TABLE(PLI_NUMBER_TABLE,
2975 P 3095 1
2976 P 3096 1 NUMBER_STATE(START_STATE,
2977 P 3097 1 NUMBER_TRANSITION(DIGIT, GO_PAST_PACK, ACCUM_INT),
2978 P 3098 1 NUMBER_TRANSITION(DOT, DO_NOTHING, LEADING_DOT),
2979 P 3099 1 NUMBER_TRANSITION(OTHER, NOT_NUMBER, END_STATE)),
2980 P 3100 1
2981 P 3101 1 NUMBER_STATE(LEADING_DOT,
2982 P 3102 1 NUMBER_TRANSITION(DIGIT, GO_PAST_PACK_FRAC, ACCUM_FRAC),
2983 P 3103 1 NUMBER_TRANSITION(OTHER, NOT_NUMBER, END_STATE)),
2984 P 3104 1
2985 P 3105 1 NUMBER_STATE(ACCUM_INT,
```

```

2986 P 3106 1 NUMBER_TRANSITION(DIGIT, GO_PAST_PACK, ACCUM_INT),
2987 P 3107 1 NUMBER_TRANSITION(DOT, DO_NOTHING, ACCUM_FRAC),
2988 P 3108 1 NUMBER_TRANSITION(HEXDIGIT, DO_NOTHING, ACCUM_HEX),
2989 P 3109 1 NUMBER_TRANSITION(B, DO_NOTHING, ACCUM_HEX),
2990 P 3110 1 NUMBER_TRANSITION(D, DO_NOTHING, ACCUM_HEX),
2991 P 3111 1 NUMBER_TRANSITION(E, DO_NOTHING, ACCUM_HEX),
2992 P 3112 1 NUMBER_TRANSITION(OTHER, GOT_PACK_NUMBER, END_STATE)),
2993 P 3113 1
2994 P 3114 1 NUMBER_STATE(ACCUM_HEX,
2995 P 3115 1 NUMBER_TRANSITION(DIGIT, DO_NOTHING, ACCUM_HEX),
2996 P 3116 1 NUMBER_TRANSITION(HEXDIGIT, DO_NOTHING, ACCUM_HEX),
2997 P 3117 1 NUMBER_TRANSITION(B, DO_NOTHING, ACCUM_HEX),
2998 P 3118 1 NUMBER_TRANSITION(D, DO_NOTHING, ACCUM_HEX),
2999 P 3119 1 NUMBER_TRANSITION(E, DO_NOTHING, ACCUM_HEX),
3000 P 3120 1 NUMBER_TRANSITION(OTHER, GOT_NUMBER, END_STATE)),
3001 P 3121 1
3002 P 3122 1 NUMBER_STATE(ACCUM_FRAC,
3003 P 3123 1 NUMBER_TRANSITION(DIGIT, GO_PAST_PACK_FRAC, ACCUM_FRAC),
3004 P 3124 1 NUMBER_TRANSITION(DOT, BACKUP_PTRS, END_STATE),
3005 P 3125 1 NUMBER_TRANSITION(E, MARK_E_EXP, GET_EXPONENT),
3006 P 3126 1 NUMBER_TRANSITION(D, MARK_D_EXP, GET_EXPONENT),
3007 P 3127 1 NUMBER_TRANSITION(G, MARK_G_EXP, GET_EXPONENT),
3008 P 3128 1 NUMBER_TRANSITION(Q, MARK_Q_EXP, GET_EXPONENT),
3009 P 3129 1 NUMBER_TRANSITION(OTHER, GOT_PACK_NUMBER, END_STATE)),
3010 P 3130 1
3011 P 3131 1 NUMBER_STATE(GET_EXPONENT,
3012 P 3132 1 NUMBER_TRANSITION(DIGIT, DO_NOTHING, ACCUM_EXP),
3013 P 3133 1 NUMBER_TRANSITION(PLUS, DO_NOTHING, GET_EXP_SIGN),
3014 P 3134 1 NUMBER_TRANSITION(MINUS, DO_NOTHING, GET_EXP_SIGN),
3015 P 3135 1 NUMBER_TRANSITION(OTHER, BACKUP_PTRS, END_STATE)),
3016 P 3136 1
3017 P 3137 1 NUMBER_STATE(GET_EXP_SIGN,
3018 P 3138 1 NUMBER_TRANSITION(DIGIT, DO_NOTHING, ACCUM_EXP),
3019 P 3139 1 NUMBER_TRANSITION(OTHER, GOT_NUMBER, END_STATE)),
3020 P 3140 1
3021 P 3141 1 NUMBER_STATE(ACCUM_EXP,
3022 P 3142 1 NUMBER_TRANSITION(DIGIT, DO_NOTHING, ACCUM_EXP),
3023 P 3143 1 NUMBER_TRANSITION(OTHER, GOT_NUMBER, END_STATE)),
3024 P 3144 1
3025 P 3145 1 NUMBER_STATE(END_STATE,
3026 P 3146 1 NUMBER_TRANSITION(OTHER, GIVE_ERROR, END_STATE)));
3027 P 3147 1
3028 P 3148 1
3029 P 3149 1 ! Define the Primary Parser State Table for language PL/I. Each transition
3030 P 3150 1 ! Entry in the state table has this format:
3031 P 3151 1
3032 P 3152 1 PRIMARY_TRANSITION(operator-code, action, next-state)
3033 P 3153 1
3034 P 3154 1 ! where the first parameter is the operator code which causes the transition
3035 P 3155 1 ! to be taken, the second parameter is the action routine CASE index for the
3036 P 3156 1 ! transition, and the third parameter is the next state in the Finite-State
3037 P 3157 1 ! Machine.
3038 P 3158 1
3039 P 3159 1 PRIMARY_STATE_TABLE(PLI_PRIMARY_TABLE,
3040 P 3160 1
3041 P 3161 1 PRIMARY_STATE(START_STATE,
3042 P 3162 1 PRIMARY_TRANSITION(GLOBAL_SLASH, START_GBL, GET_GLOBAL),

```

```

3043 P 3163 1 PRIMARY_TRANSITION(BACKSLASH, START SLASH, GOT BACKSLASH),
3044 P 3164 1 PRIMARY_TRANSITION(INVOCNUM, SLASH INVOCNUM, GOT BACKSLASH),
3045 P 3165 1 PRIMARY_TRANSITION(DOT, START DOT PLI, GOT DOT),
3046 P 3166 1 PRIMARY_TRANSITION(SUBSCRIPT, START SUBSCR-PLI, GOT SUBSCRIPT),
3047 P 3167 1 PRIMARY_TRANSITION(PLI Deref, START Deref PLI, START STATE),
3048 P 3168 1 PRIMARY_TRANSITION(PRIMARY_TERM, START_TERM, END_STATE)),
3049 P 3169 1
3050 P 3170 1 PRIMARY_STATE(GET GLOBAL,
3051 P 3171 1 PRIMARY_TRANSITION(PRIMARY_TERM, GBL_TERM, END_STATE)),
3052 P 3172 1
3053 P 3173 1 PRIMARY_STATE(GOT BACKSLASH,
3054 P 3174 1 PRIMARY_TRANSITION(BACKSLASH, SLASH SLASH, GOT BACKSLASH),
3055 P 3175 1 PRIMARY_TRANSITION(INVOCNUM, SLASH INVOCNUM, GOT BACKSLASH),
3056 P 3176 1 PRIMARY_TRANSITION(DOT, SLASH DOT PLI, GOT DOT),
3057 P 3177 1 PRIMARY_TRANSITION(SUBSCRIPT, SLASH SUBSCR-PLI, GOT SUBSCRIPT),
3058 P 3178 1 PRIMARY_TRANSITION(PLI Deref, SLASH Deref PLI, START STATE),
3059 P 3179 1 PRIMARY_TRANSITION(PRIMARY_TERM, SLASH_TERM, END_STATE)),
3060 P 3180 1
3061 P 3181 1 PRIMARY_STATE(GOT DOT,
3062 P 3182 1 PRIMARY_TRANSITION(DOT, DOT DOT PLI, GOT DOT),
3063 P 3183 1 PRIMARY_TRANSITION(SUBSCRIPT, DOT SUBSCR-PLI, GOT SUBSCRIPT),
3064 P 3184 1 PRIMARY_TRANSITION(PLI Deref, DOT Deref PLI, START STATE),
3065 P 3185 1 PRIMARY_TRANSITION(PRIMARY_TERM, DOT_TERM_PLI, END_STATE)),
3066 P 3186 1
3067 P 3187 1 PRIMARY_STATE(GOT SUBSCRIPT,
3068 P 3188 1 PRIMARY_TRANSITION(DOT, SUBSCR DOT PLI, GOT DOT),
3069 P 3189 1 PRIMARY_TRANSITION(SUBSCRIPT, SUBSCR SUBSCR-PLI, GOT SUBSCRIPT),
3070 P 3190 1 PRIMARY_TRANSITION(PLI Deref, SUBSCR Deref PLI, START STATE),
3071 P 3191 1 PRIMARY_TRANSITION(PRIMARY_TERM, SUBSCR_TERM_PLI, END_STATE)),
3072 P 3192 1
3073 P 3193 1 PRIMARY_STATE(END_STATE));
3074 P 3194 1
3075 P 3195 1
3076 P 3196 1 ! Define the table of pointers to the parse tables for PL/I.
3077 P 3197 1 !
3078 P 3198 1 LANGUAGE_TABLES(LANGUAGE = PLI,
3079 P 3199 1 CHARTBL = PLI_CHARTBL,
3080 P 3200 1 IDENT OPTBL = PLI_IDENT OPTBL,
3081 P 3201 1 OPCHAR OPTBL = PLI_OPCHAR OPTBL,
3082 P 3202 1 NUMBER_TABLE = PLI_NUMBER_TABLE,
3083 P 3203 1 PRIMARY_TABLE = PLI_PRIMARY_TABLE,
3084 P 3204 1 SUBSCR_TERMS = PLI_SUBSCR_TERM_TBL,
3085 P 3205 1 PRIDTBL = PLI_PRID_TABLE,
3086 P 3206 1 BIF_TABLE = PLI_FUNCTION_TABLE,
3087 P 3207 1 MULTIPLE SUBSCR = FALSE,
3088 P 3208 1 COMPONENTS_IN_PATHNAME = TRUE);
```


R P G P A R S E T A B L E S

This section includes all the Lexical Scanner and Parser tables needed to scan and parse the RPG language.

Define the RPG Character Table. What is listed here is actually a list of exceptions to the Character Table for Language UNKNOWN.

```
CHAR_EXCEPTION TABLE(RPG CHARTBL,  
  CHAR_ENTRY('*', OTHER, NOTHING),  
  CHAR_ENTRY('<', OTHER, OPCHAR, OPCHAR_INFIX, ADDRESS_OP, TERMINATOR),  
  CHAR_ENTRY('>', OTHER, OPCHAR, OPCHAR_INFIX, TERMINATOR),  
  CHAR_ENTRY('=', OTHER, OPCHAR, OPCHAR_INFIX, TERMINATOR),  
  CHAR_ENTRY('#', OTHER, IDENT_ANYWHERE),  
  CHAR_ENTRY(' ', OTHER, IDENT_ANYWHERE),  
  CHAR_ENTRY('$', OTHER, IDENT_ANYWHERE));
```

Define the RPG Operator Table for operators whose names are identifiers.

```
OPERATOR TABLE(RPG IDENT_OPTBL,  
  OPERATOR_ENTRY('NOT', NOT, PREFIX, 200, 11),  
  OPERATOR_ENTRY('AND', AND, INFIX, 10, 10),  
  OPERATOR_ENTRY('OR', OR, INFIX, 10, 10),  
  OPERATOR_ENTRY('NOT', INFIX_NOT, INFIX, 15, 15, LEXICAL));
```

BIND

```
RPG_NOT_EQL_TOKEN =  
  OPERATOR_ENTRY('NOT =', NOT_EQUAL, INFIX, 15, 15),  
RPG_NOT_GTR_TOKEN =  
  OPERATOR_ENTRY('NOT >', LSS_EQUAL, INFIX, 15, 15),  
RPG_NOT_LSS_TOKEN =  
  OPERATOR_ENTRY('NOT <', GTR_EQUAL, INFIX, 15, 15);
```

Define the RPG Operator Table for operators whose names are composed of operator characters such as '+', '-', or '*'. This table includes operators which are part of DEBUG Primary Symbols (such as '\').

```
OPERATOR TABLE(RPG OPCHAR_OPTBL,  
  OPERATOR_ENTRY('\', GLOBAL_SLASH, PREFIX, 0, 0, PRIMARY),  
  OPERATOR_ENTRY('\', BACKSLASH, INFIX, 0, 0, PRIMARY),  
  OPERATOR_ENTRY('(', SUBSCRIPT, POSTFIX, 0, 0, PRIMARY),  
  OPERATOR_ENTRY('>', PREFIX_GTR, PREFIX, 200, 15, LEXICAL),  
  OPERATOR_ENTRY('<', PREFIX_LSS, PREFIX, 200, 15, LEXICAL),  
  OPERATOR_ENTRY('=', PREFIX_EQL, PREFIX, 200, 15, LEXICAL),  
  OPERATOR_ENTRY('(', OPENPAREN, PREFIX, 200, 5, LEXICAL),  
  OPERATOR_ENTRY(')', CLOSEPAREN, POSTFIX, 6, 200, LEXICAL),  
  OPERATOR_ENTRY('/', DIVIDE, INFIX, 30, 30),  
  OPERATOR_ENTRY('+', UNARY_PLUS, PREFIX, 200, 25),  
  OPERATOR_ENTRY('-', UNARY_MINUS, PREFIX, 200, 25),  
  OPERATOR_ENTRY('+', ADD, INFIX, 20, 20),  
  OPERATOR_ENTRY('-', SUBTRACT, INFIX, 20, 20),  
  OPERATOR_ENTRY('>', GTR_THAN, INFIX, 15, 15);
```

```
3147 P 3266 1 OPERATOR_ENTRY('<', LSS_THAN, INFIX, 15, 15);
3148 3267 1 OPERATOR_ENTRY('= ', EQUAL, INFIX, 15, 15);
3149 3268 1
3150 3269 1 BIND
3151 3270 1 RPG_MULTIPLY_TOKEN =
3152 3271 1 OPERATOR_ENTRY('* ', MULTIPLY, INFIX, 30, 30);
3153 3272 1
3154 3273 1
3155 3274 1 ! Define the RPG Terminator Lexical Token Table for subscript expressions.
3156 3275 1
3157 P 3276 1 TERMINATOR_TABLE(RPG_SUBSCR_TERM_TBL,
3158 P 3277 1 TERMINATOR_ENTRY(')', TERM_CLOSE),
3159 P 3278 1 TERMINATOR_ENTRY(':', TERM_COLON, MUST_BE_SINGLE),
3160 3279 1 TERMINATOR_ENTRY(',', TERM_COMMA));
3161 3280 1
3162 3281 1
3163 3282 1 ! Define the RPG Predefined Identifier Table.
3164 3283 1
3165 3284 1 PRID_TABLE(RPG_PRID_TABLE);
3166 3285 1
3167 3286 1
3168 3287 1 ! Define the RPG Built-in Function Table.
3169 3288 1
3170 3289 1 BUILT_IN_FUNCTION_TABLE(RPG_FUNCTION_TABLE);
3171 3290 1
3172 3291 1
3173 3292 1 ! Define the RPG Number Scanner State Table. This table defines the states
3174 3293 1 of a Finite-State Machine which picks up all valid numeric constants in the
3175 3294 1 language. Each Transition Entry is of the form:
3176 3295 1
3177 3296 1 NUMBER_TRANSITION(character-class, action-index, next-state)
3178 3297 1
3179 3298 1 where the character-class and action-index names are automatically prefixed
3180 3299 1 by 'NUMSTSK_CLASS_' or 'NUMSTSK_ACT_' by the NUMBER_TRANSITION macro.
3181 3300 1
3182 3301 1 BIND
3183 3302 1 RPG_NUMBER_TABLE = PLI_NUMBER_TABLE;
3184 3303 1
3185 3304 1
3186 3305 1 ! Define the Primary Parser State Table for language RPG. Each transition
3187 3306 1 Entry in the state table has this format:
3188 3307 1
3189 3308 1 PRIMARY_TRANSITION(operator-code, action, next-state)
3190 3309 1
3191 3310 1 where the first parameter is the operator code which causes the transition
3192 3311 1 to be taken, the second parameter is the action routine (ASE index for the
3193 3312 1 transition, and the third parameter is the next state in the Finite-State
3194 3313 1 Machine.
3195 3314 1
3196 P 3315 1 PRIMARY_STATE_TABLE(RPG_PRIMARY_TABLE,
3197 P 3316 1
3198 P 3317 1 PRIMARY_STATE(START_STATE,
3199 P 3318 1 PRIMARY_TRANSITION(GLOBAL_SLASH, START_GBL, GET_GLOBAL),
3200 P 3319 1 PRIMARY_TRANSITION(BACKSLASH, START_SLASH, GOT_BACKSLASH),
3201 P 3320 1 PRIMARY_TRANSITION(INVOCNUM, SLASH_INVOCNUM, GOT_BACKSLASH),
3202 P 3321 1 PRIMARY_TRANSITION(SUBSCRIPT, START_SUBSCR, GOT_SUBSCRIPT),
3203 P 3322 1 PRIMARY_TRANSITION(PRIMARY_TERM, START_TERM, END_STATE)),
```

```

: 3204      P 3323 1
: 3205      P 3324 1
: 3206      P 3325 1
: 3207      P 3326 1
: 3208      P 3327 1
: 3209      P 3328 1
: 3210      P 3329 1
: 3211      P 3330 1
: 3212      P 3331 1
: 3213      P 3332 1
: 3214      P 3333 1
: 3215      P 3334 1
: 3216      P 3335 1
: 3217      P 3336 1
: 3218      P 3337 1
: 3219      P 3338 1
: 3220      P 3339 1
: 3221      P 3340 1
: 3222      P 3341 1
: 3223      P 3342 1
: 3224      P 3343 1
: 3225      P 3344 1
: 3226      P 3345 1
: 3227      P 3346 1
: 3228      P 3347 1
: 3229      P 3348 1
: 3230      P 3349 1
: 3231      P 3350 1

PRIMARY STATE(GET GLOBAL,
PRIMARY_TRANSITION(PRIMARY_TERM, GBL_TERM, END_STATE)),

PRIMARY STATE(GOT BACKSLASH,
PRIMARY_TRANSITION(BACKSLASH, SLASH SLASH, GOT BACKSLASH),
PRIMARY_TRANSITION(INVOCNUM, SLASH INVOCNUM, GOT BACKSLASH),
PRIMARY_TRANSITION(SUBSCRIPT, SLASH SUBSCR, GOT SUBSCRIPT),
PRIMARY_TRANSITION(PRIMARY_TERM, SLASH_TERM, END_STATE)),

PRIMARY STATE(GOT SUBSCRIPT,
PRIMARY_TRANSITION(SUBSCRIPT, SUBSCR SUBSCR, GOT SUBSCRIPT),
PRIMARY_TRANSITION(PRIMARY_TERM, SUBSCR_TERM, END_STATE)),

PRIMARY_STATE(END_STATE));

! Define the table of pointers to the parse tables for RPG.
LANGUAGE_TABLES(LANGUAGE = RPG,
CHARTBL = RPG_CHARTBL,
IDENT_OPTBL = RPG_IDENT_OPTBL,
OPCHAR_OPTBL = RPG_OPCHAR_OPTBL,
NUMBER_TABLE = RPG_NUMBER_TABLE,
PRIMARY_TABLE = RPG_PRIMARY_TABLE,
SUBSCR_TERMS = RPG_SUBSCR_TERM_TBL,
PRIDTBL = RPG_PRID_TABLE,
BIF_TABLE = PASCAL_FUNCTION_TABLE);

```



```
3233 3351 1  TABLE OF POINTERS TO LANGUAGE TABLES
3234 3352 1
3235 3353 1
3236 3354 1
3237 3355 1 This section contains the table of pointers to the language-specific
3238 3356 1 tables of parser table pointers. In other words, this table is indexed
3239 3357 1 by language code to yield a pointer (relative to TABLEBASE) to the
3240 3358 1 table of table pointers built by the LANGUAGE_TABLES macro for the
3241 3359 1 individual language above.
3242 3360 1
3243 3361 1
3244 3362 1 Define the table of pointers to language-specific tables.
3245 3363 1
3246 3364 1 OWN
3247 3365 1 LANGUAGE_TABLE_PTRS:
3248 3366 1 VECTOR[DBG$K_MAX_LANGUAGE + 1, LONG] PSECT(DBG$PLIT) PRESET(
3249 3367 1
3250 3368 1 [DBG$K_MACRO] = MACRO TABLES - TABLEBASE, ! MACRO
3251 3369 1 [DBG$K_FORTRAN] = FORTRAN TABLES - TABLEBASE, ! FORTRAN
3252 3370 1 [DBG$K_BLISS] = BLISS TABLES - TABLEBASE, ! BLISS
3253 3371 1 [DBG$K_COBOL] = COBOL TABLES - TABLEBASE, ! COBOL
3254 3372 1 [DBG$K_BASIC] = BASIC TABLES - TABLEBASE, ! BASIC
3255 3373 1 [DBG$K_PLI] = PLI TABLES - TABLEBASE, ! PL/I
3256 3374 1 [DBG$K_PASCAL] = PASCAL TABLES - TABLEBASE, ! PASCAL
3257 3375 1 [DBG$K_C] = C TABLES - TABLEBASE, ! C
3258 3376 1 [DBG$K_RPG] = RPG TABLES - TABLEBASE, ! RPG
3259 3377 1 [DBG$K_ADA] = ADA TABLES - TABLEBASE, ! ADA
3260 3378 1 [DBG$K_UNKNOWN] = UNKNOWN TABLES - TABLEBASE, ! UNKNOWN
```

```
.. 3262      3379 1 ROUTINE AAA_DUMMY: NOVALUE =
.. 3263      3380 1
.. 3264      3381 1 FUNCTION
.. 3265      3382 1 This is a dummy routine which is never called. Its sole purpose is
.. 3266      3383 1 to force out the machine listing for all the above data tables before
.. 3267      3384 1 the source listing for the first real routine in the module.
.. 3268      3385 1
.. 3269      3386 1 INPUTS
.. 3270      3387 1 NONE
.. 3271      3388 1
.. 3272      3389 1 OUTPUTS
.. 3273      3390 1 NONE
.. 3274      3391 1
.. 3275      3392 1
.. 3276      3393 2 BEGIN
.. 3277      3394 2
.. 3278      3395 2 RETURN;
.. 3279      3396 2
.. 3280      3397 1 END;
```

										.TITLE	DBGPARSER	
										.IDENT	\V04-000\	
										.PSECT	DBG\$PLIT,NOWRT, SHR, PIC,0	
						00	03	00000	P.AAA:	.BYTE	3, 0	
							0033	00002		.WORD	51	
	00	00	00	00	00	00	00	00004		.BYTE	0, 0, 0, 0, 0, 0, 0, 0	
	54	52	45	56	4E	4F	43	07	0000C	.ASCII	<7>\CONVERT\	
							00	03	00014	P.AAB:	.BYTE	3, 0
								0032	00016	.WORD	50	
	00	00	00	00	00	00	00	00018		.BYTE	0, 0, 0, 0, 0, 0, 0, 0	
	54	49	53	4F	50	45	44	07	00020	.ASCII	<7>\DEPOSIT\	
							00	02	00028	P.AAC:	.BYTE	2, 0
								0038	0002A	.WORD	56	
59	00	00	00	00	00	00	00	0002C		.BYTE	0, 0, 0, 0, 0, 0, 0, 0	
	54	49	54	4E	45	44	49	08	00034	.ASCII	<8>\IDENTIFY\	
							00	02	0003D	P.AAD:	.BYTE	2, 0
								0042	0003F	.WORD	66	
54	00	00	00	00	00	00	00	00041		.BYTE	0, 0, 0, 0, 0, 0, 0, 0	
	53	4E	4F	43	47	45	4E	08	00049	.ASCII	<8>\NEGCONST\	
							00	02	00052	P.AAE:	.BYTE	2, 0
								0043	00054	.WORD	67	
54	00	00	00	00	00	00	00	00056		.BYTE	0, 0, 0, 0, 0, 0, 0, 0	
	53	4E	4F	43	53	4F	50	08	0005E	.ASCII	<8>\POSCONST\	
							00	02	00067	P.AAF:	.BYTE	2, 0
								0005	00069	.WORD	55	
	00	00	00	00	00	00	00	0006B		.BYTE	0, 0, 0, 0, 0, 0, 0, 0	
							2D	01	00073	.ASCII	<1>\-\	
							00	02	00075	P.AAG:	.BYTE	2, 0
								0004	00077	.WORD	4	
	00	00	00	00	00	00	00	00079		.BYTE	0, 0, 0, 0, 0, 0, 0, 0	
							2B	01	00081	.ASCII	<1>\+\	
				45	53	41	42	00083	P.AAH:	.ASCII	\BASE\	
							01	00087	P.AAJ:	.BYTE	1	
	45	4E	49	4C	25	05		00088		.ASCII	<5>\XLINE\	

				01	0008E	P.AAK:	.BYTE	1	
4E	49	4C	25	04	0008F		.ASCII	<4>\XLIN\	
				01	00094	P.AAL:	.BYTE	1	
	49	4C	25	03	00095		.ASCII	<3>\XLI\	
				02	00099	P.AAM:	.BYTE	2	
4C	45	42	41	4C	25	06	0009A	.ASCII	<6>\XLABEL\
				02	000A1	P.AAN:	.BYTE	2	
45	42	41	4C	25	05	000A2	.ASCII	<5>\XLABEL\	
				02	000A8	P.AAO:	.BYTE	2	
42	41	4C	25	04	000A9		.ASCII	<4>\XLAB\	
				02	000AE	P.AAP:	.BYTE	2	
	41	4C	25	03	000AF		.ASCII	<3>\XLA\	
				03	000B3	P.AAQ:	.BYTE	3	
45	4D	41	4E	25	05	000B4	.ASCII	<5>\XNAME\	
				08	000BA	P.AAR:	.BYTE	8	
	30	52	25	03	000BB		.ASCII	<3>\XR0\	
				08	000BF	P.AAS:	.BYTE	8	
	31	52	25	03	000C0		.ASCII	<3>\XR1\	
				08	000C4	P.AAT:	.BYTE	8	
	32	52	25	03	000C5		.ASCII	<3>\XR2\	
				08	000C9	P.AAU:	.BYTE	8	
	33	52	25	03	000CA		.ASCII	<3>\XR3\	
				08	000CE	P.AAV:	.BYTE	8	
	34	52	25	03	000CF		.ASCII	<3>\XR4\	
				08	000D3	P.AAW:	.BYTE	8	
	35	52	25	03	000D4		.ASCII	<3>\XR5\	
				08	000D8	P.AAX:	.BYTE	8	
	36	52	25	03	000D9		.ASCII	<3>\XR6\	
				08	000DD	P.AAY:	.BYTE	8	
	37	52	25	03	000DE		.ASCII	<3>\XR7\	
				08	000E2	P.AAZ:	.BYTE	8	
	38	52	25	03	000E3		.ASCII	<3>\XR8\	
				08	000E7	P.ABA:	.BYTE	8	
	39	52	25	03	000E8		.ASCII	<3>\XR9\	
				08	000EC	P.ABB:	.BYTE	8	
30	31	52	25	04	000ED		.ASCII	<4>\XR10\	
				08	000F2	P.ABC:	.BYTE	8	
31	31	52	25	04	000F3		.ASCII	<4>\XR11\	
				08	000F8	P.ABD:	.BYTE	8	
32	31	52	25	04	000F9		.ASCII	<4>\XR12\	
				08	000FE	P.ABE:	.BYTE	8	
33	31	52	25	04	000FF		.ASCII	<4>\XR13\	
				08	00104	P.ABF:	.BYTE	8	
34	31	52	25	04	00105		.ASCII	<4>\XR14\	
				08	0010A	P.ABG:	.BYTE	8	
35	31	52	25	04	0010B		.ASCII	<4>\XR15\	
				08	00110	P.ABH:	.BYTE	8	
	50	41	25	03	00111		.ASCII	<3>\XAP\	
				08	00115	P.ABI:	.BYTE	8	
	50	46	25	03	00116		.ASCII	<3>\XFP\	
				08	0011A	P.ABJ:	.BYTE	8	
	50	53	25	03	0011B		.ASCII	<3>\XSP\	
				08	0011F	P.ABK:	.BYTE	8	
	43	50	25	03	00120		.ASCII	<3>\XPC\	
				08	00124	P.ABL:	.BYTE	8	
4C	53	50	25	04	00125		.ASCII	<4>\XPSL\	
				08	0012A	P.ABM:	.BYTE	8	

30	72	25	03	0012B		.ASCII	<3>\%r0\				
			08	0012F	P.ABN:	.BYTE	8				
31	72	25	03	00130		.ASCII	<3>\%r1\				
			08	00134	P.ABO:	.BYTE	8				
32	72	25	03	00135		.ASCII	<3>\%r2\				
			08	00139	P.ABP:	.BYTE	8				
33	72	25	03	0013A		.ASCII	<3>\%r3\				
			08	0013E	P.ABQ:	.BYTE	8				
34	72	25	03	0013F		.ASCII	<3>\%r4\				
			08	00143	P.ABR:	.BYTE	8				
35	72	25	03	00144		.ASCII	<3>\%r5\				
			08	00148	P.ABS:	.BYTE	8				
36	72	25	03	00149		.ASCII	<3>\%r6\				
			08	0014D	P.ABT:	.BYTE	8				
37	72	25	03	0014E		.ASCII	<3>\%r7\				
			08	00152	P.ABU:	.BYTE	8				
38	72	25	03	00153		.ASCII	<3>\%r8\				
			08	00157	P.ABV:	.BYTE	8				
39	72	25	03	00158		.ASCII	<3>\%r9\				
			08	0015C	P.ABW:	.BYTE	8				
30	31	72	25	04	0015D	.ASCII	<4>\%r10\				
			08	00162	P.ABX:	.BYTE	8				
31	31	72	25	04	00163	.ASCII	<4>\%r11\				
			08	00168	P.ABY:	.BYTE	8				
32	31	72	25	04	00169	.ASCII	<4>\%r12\				
			08	0016E	P.ABZ:	.BYTE	8				
33	31	72	25	04	0016F	.ASCII	<4>\%r13\				
			08	00174	P.ACA:	.BYTE	8				
34	31	72	25	04	00175	.ASCII	<4>\%r14\				
			08	0017A	P.ACB:	.BYTE	8				
35	31	72	25	04	0017B	.ASCII	<4>\%r15\				
			08	00180	P.ACC:	.BYTE	8				
	70	61	25	03	00181	.ASCII	<3>\%ap\				
			08	00185	P.ACD:	.BYTE	8				
	70	66	25	03	00186	.ASCII	<3>\%fp\				
			08	0018A	P.ACE:	.BYTE	8				
	70	73	25	03	0018B	.ASCII	<3>\%sp\				
			08	0018F	P.ACF:	.BYTE	8				
	63	70	25	03	00190	.ASCII	<3>\%pc\				
			08	00194	P.ACG:	.BYTE	8				
6C	73	70	25	04	00195	.ASCII	<4>\%psl\				
			04	0019A	P.ACH:	.BYTE	4				
43	45	44	25	04	0019B	.ASCII	<4>\%DEC\				
			05	001A0	P.ACI:	.BYTE	5				
58	45	48	25	04	001A1	.ASCII	<4>\%HEX\				
			06	001A6	P.ACJ:	.BYTE	6				
54	43	4F	25	04	001A7	.ASCII	<4>\%OCT\				
			07	001AC	P.ACK:	.BYTE	7				
4E	49	42	25	04	001AD	.ASCII	<4>\%BIN\				
			08	001B2	P.ACL:	.BYTE	8				
43	4F	4C	52	55	43	25	07	001B3	.ASCII	<7>\%CURLOC\	
							08	001BB	P.ACM:	.BYTE	8
4C	41	56	52	55	43	25	07	001BC	.ASCII	<7>\%CURVAL\	
							08	001C4	P.ACN:	.BYTE	8
43	4F	4C	56	45	52	50	25	08	001C5	.ASCII	<8>\%PREVLOC\
							08	001CE	P.ACO:	.BYTE	8
43	4F	4C	54	58	45	4E	25	08	001CF	.ASCII	<8>\%NEXTLOC\

	54	4E	43	52	41	50	25	08			
00000025	0000001E	00000016	00000011	0000000B	00000004	0000003B	00000004	001D8	P.ACP:	.BYTE	8
00000046	00000041	0000003C	00000037	00000030	0000002B	0000002B	0000002B	001D9		.ASCII	<7>\%PARCNT\
00000064	0000005F	0000005A	00000055	00000050	0000004B	0000004B	0000004B	001E1		.BLKB	3
00000087	00000081	0000007B	00000075	0000006F	00000069	00000069	00000069	001E4		.LONG	59
000000A7	000000A1	0000009C	00000097	00000092	0000008D	0000008D	0000008D	001E8	P.AAI:	.LONG	4, 11, 17, 22, 30, 37, 43, 48, 55, 60, -
000000C5	000000C0	000000BB	000000B6	000000B1	000000AC	000000AC	000000AC	00200			65, 70, 75, 80, 85, 90, 95, 100, 105, -
000000E5	000000DF	000000D9	000000D4	000000CF	000000CA	000000CA	000000CA	00218			111, 117, 123, 129, 135, 141, 146, 151, -
00000107	00000102	000000FD	000000F7	000000F1	000000EB	000000EB	000000EB	00230			156, 161, 167, 172, 177, 182, 187, 192, -
00000129	00000123	0000011D	00000117	00000111	0000010C	0000010C	0000010C	00248			197, 202, 207, 212, 217, 223, 229, 235, -
	00000155	0000014B	00000141	00000138	0000012F	0000012F	0000012F	00260			241, 247, 253, 258, 263, 268, 273, 279, -
								00 02			285, 291, 297, 303, 312, 321, 331, 341
								0001	P.ACQ:	.BYTE	2, 0
65 72 70 78 65 20 66	00 00 00	00 00 00	00 00 00	00 00 00	00 00 00	00 00 00	00 00 00	002D4		.WORD	1
	6F 20 74	72 61 74	73 13	002D6				002D8		.BYTE	1, -56, 0, 0, 0, 0, 0, 0
		6E 6F 69	73 73	002E0				002EF		.ASCII	<19>\start of expression\
				002F4				002F6	P.ACR:	.BYTE	4, 2
73 73 65 72 70 78 65	00 00 00	00 00 00	00 00 00	00 00 00	00 00 00	00 00 00	00 00 00	002F8		.WORD	2
	20 66 6F	20 64 6E	65 11	00300				0030F		.BYTE	-56, 2, 0, 0, 0, 0, 0, 0
				00312				00314		.ASCII	<17>\end of expression\
				00316				0031E	P.ACS:	.BYTE	4, 1
				0032C				0032E		.WORD	1
6C 6F 62 6D 79 73	00 00 00	00 00 00	00 00 00	00 00 00	00 00 00	00 00 00	00 00 00	00330		.BYTE	0, 0, 0, 0, 0, 0, 0, 0
	20 66 6F	20 64 6E	65 0D	00338				0033D		.ASCII	<13>\end of symbol\
				00341				00349	P.ACT:	.BYTE	2, 2
				00350				0035A		.WORD	52
				0035F				00363		.BYTE	-66, -56, 0, 0, 0, 0, 0, 0
				0036B				00370		.ASCII	<4>\%DEC\
				00372				00378	P.ACU:	.BYTE	2, 2
				00380				00384		.WORD	53
				00388				00392		.BYTE	-66, -56, 0, 0, 0, 0, 0, 0
				00394				00398		.ASCII	<4>\%HEX\
				003A1					P.ACV:	.BYTE	2, 2
										.WORD	54
										.BYTE	-66, -56, 0, 0, 0, 0, 0, 0
										.ASCII	<4>\%OCT\
									P.ACW:	.BYTE	2, 2
										.WORD	55
										.BYTE	-66, -56, 0, 0, 0, 0, 0, 0
										.ASCII	<4>\%BIN\
									P.ACX:	.BYTE	1, 0
										.WORD	1
										.BYTE	0, 0, 0, 0
										.ASCII	<7>\%CURLOC\
									P.ACY:	.BYTE	1, 0
										.WORD	1
										.BYTE	0, 0, 0, 0
										.ASCII	<7>\%CURVAL\
									P.ACZ:	.BYTE	1, 0
										.WORD	1
										.BYTE	0, 0, 0, 0
										.ASCII	<8>\%PREVLOC\
									P.ADB:	.BYTE	2, 0

00	00	00	00	00	00	0003	003A3	.WORD	3	
						C8 28	003A5	.BYTE	40	-56, 0, 0, 0, 0, 0, 0
						2E 01	003AD	.ASCII	<1>\.	
						00 02	003AF	P.ADC: .BYTE	2	0
						0003	003B1	.WORD	3	
00	00	00	00	00	00	C8 28	003B3	.BYTE	40	-56, 0, 0, 0, 0, 0, 0
						40 01	003BB	.ASCII	<1>\a\	
						00 03	003BD	P.ADD: .BYTE	3	0
						0006	003BF	.WORD	6	
00	00	00	00	00	00	0A 0A	003C1	.BYTE	10	10, 0, 0, 0, 0, 0, 0
						28 01	003C9	.ASCII	<1>\+\	
						00 03	003CB	P.ADE: .BYTE	3	0
						0007	003CD	.WORD	7	
00	00	00	00	00	00	0A 0A	003CF	.BYTE	10	10, 0, 0, 0, 0, 0, 0
						2D 01	003D7	.ASCII	<1>\-\	
						00 02	003D9	P.ADF: .BYTE	2	0
						0004	003DB	.WORD	4	
00	00	00	00	00	00	C8 14	003DD	.BYTE	20	-56, 0, 0, 0, 0, 0, 0
						2B 01	003E5	.ASCII	<1>\+\	
						00 02	003E7	P.ADG: .BYTE	2	0
						0005	003E9	.WORD	5	
00	00	00	00	00	00	C8 14	003EB	.BYTE	20	-56, 0, 0, 0, 0, 0, 0
						2D 01	003F3	.ASCII	<1>\-\	
						00 03	003F5	P.ADH: .BYTE	3	0
						0008	003F7	.WORD	8	
00	00	00	00	00	00	1E 1E	003F9	.BYTE	30	30, 0, 0, 0, 0, 0, 0
						2A 01	00401	.ASCII	<1>*\	
						00 03	00403	P.ADI: .BYTE	3	0
						0009	00405	.WORD	9	
00	00	00	00	00	00	1E 1E	00407	.BYTE	30	30, 0, 0, 0, 0, 0, 0
						2F 01	0040F	.ASCII	<1>\/\	
						02 04	00411	P.ADJ: .BYTE	4	2
						0031	00413	.WORD	4	
00	00	00	00	00	00	32 C8	00415	.BYTE	-56	50, 0, 0, 0, 0, 0, 0
						3C 01	0041D	.ASCII	<1>\<\	
						02 02	0041F	P.ADK: .BYTE	2	2
						000B	00421	.WORD	11	
00	00	00	00	00	00	C8 05	00423	.BYTE	5	-56, 0, 0, 0, 0, 0, 0
						28 01	0042B	.ASCII	<1>\(\	
						02 04	0042D	P.ADL: .BYTE	4	2
						000C	0042F	.WORD	12	
00	00	00	00	00	00	06 C8	00431	.BYTE	-56	6, 0, 0, 0, 0, 0, 0
						29 01	00439	.ASCII	<1>\)\	
						0000000B	0043B	.BLKB	1	
00000364	00000356	00000348	0000033A	0000032C	0000031E	0000031E	00440	P.ADA: .LONG	798, 812, 826, 840, 854, 868, 882, 896, -	
	000003AA	0000039C	0000038E	00000380	00000372	00000372	00458		910, 924, 938	
					00000000	0046C	.LONG	0		
						00470	P.ADM: .BLKB	0		
					00 00	00470	P.ADO: .BYTE	0	0	
					0001	00472	.WORD	1		
				00 00	00 00	00474	.BYTE	0	0, 0, 0	
					2C 01	00478	.ASCII	<1>\,\		
						0047A	.BLKB	2		
					00000001	0047C	.LONG	1		
					000003ED	00480	P.ADN: .LONG	1005		
					01 00	00484	P.ADQ: .BYTE	0, 1		

		0004	00486		.WORD	4	
00	00	00 00	00488		.BYTE	0 0 0 0	
		3D 01	0048C		.ASCII	<1>\=\	
			0048E		.BLKB	2	
		00000001	00490		.LONG	1	
		00000401	00494	P.ADP:	.LONG	1025	
		01 00	00498	P.ADS:	.BYTE	0 1	
		0005	0049A		.WORD	5	
00	00	00 00	0049C		.BYTE	0 0 0 0	
		4F 44 02	004A0		.ASCII	<2>\DO\	
			004A3		.BLKB	1	
		00000001	004A4		.LONG	1	
		00000415	004A8	P.ADR:	.LONG	1045	
		01 00	004AC	P.ADU:	.BYTE	0 1	
		0006	004AE		.WORD	6	
00	00	00 00	004B0		.BYTE	0 0 0 0	
4E	45	48 54 04	004B4		.ASCII	<4>\THEN\	
			004B9		.BLKB	3	
		00000001	004BC		.LONG	1	
		00000429	004C0	P.ADT:	.LONG	1065	
		00 00	004C4	P.ADW:	.BYTE	0 0	
		0001	004C6		.WORD	1	
00	00	00 00	004C8		.BYTE	0 0 0 0	
		2C 01	004CC		.ASCII	<1>\,\	
		02 00	004CE	P.ADX:	.BYTE	0 2	
		0003	004D0		.WORD	3	
00	00	00 00	004D2		.BYTE	0 0 0 0	
		3A 01	004D6		.ASCII	<1>\:\	
		00000002	004D8		.LONG	2	
0000044B		00000441	004DC	P.ADV:	.LONG	1089, 1099	
		00 00	004E4	P.ADZ:	.BYTE	0 0	
		0001	004E6		.WORD	1	
00	00	00 00	004E8		.BYTE	0 0 0 0	
		2C 01	004EC		.ASCII	<1>\,\	
		01 00	004EE	P.AEA:	.BYTE	0 1	
		0007	004F0		.WORD	7	
00	00	00 00	004F2		.BYTE	0 0 0 0	
4E	45	48 57 04	004F6		.ASCII	<4>\WHEN\	
		01 00	004FB	P.AEB:	.BYTE	0 1	
		0005	004FD		.WORD	5	
00	00	00 00	004FF		.BYTE	0 0 0 0	
		4F 44 02	00503		.ASCII	<2>\DO\	
			00506		.BLKB	2	
		00000003	00508		.LONG	3	
00000478	0000046B	00000461	0050C	P.ADY:	.LONG	1121, 1131, 1144	
		00 00	00518	P.AED:	.BYTE	0 0	
		000B	0051A		.WORD	11	
00	00	00 00	0051C		.BYTE	0 0 0 0	
		28 01	00520		.ASCII	<1>\(\	
			00522		.BLKB	2	
		00000001	00524		.LONG	1	
		00000495	00528	P.AEC:	.LONG	1173	
		00 00	0052C	P.AEF:	.BYTE	0 0	
		0001	0052E		.WORD	1	
00	00	00 00	00530		.BYTE	0 0 0 0	
		2C 01	00534		.ASCII	<1>\,\	
		01 00	00536	P.AEG:	.BYTE	0 1	

```
00 00 00 00 0002 00538 .WORD 2
00 00 00 00 0053A .BYTE 0, 0, 0, 0
00 00 00 01 0053E .ASCII <1>\.\
000004B3 00000002 00540 .LONG 2
000004B3 000004A9 00544 P.AEE: .LONG 1193, 1203
00 00 01 00 0054C P.AEI: .BYTE 0, 1
0000 0054E .WORD 13
00 00 00 00 00550 .BYTE 0, 0, 0, 0
00 4F 54 02 00554 .ASCII <2>\fo\
00557 .BLKB 1
00000001 00558 .LONG 1
000004C9 0055C P.AEH: .LONG 1225
00 01 00 00 00560 P.AEK: .BYTE 0, 1
0000 00562 .WORD 14
00 00 00 00 00564 .BYTE 0, 0, 0, 0
00 59 42 02 00568 .ASCII <2>\BY\
00 01 00 00 0056B P.AEL: .BYTE 0, 1
0000 0056D .WORD 5
00 00 00 00 0056F .BYTE 0, 0, 0, 0
00 4F 44 02 00573 .ASCII <2>\DO\
00576 .BLKB 2
00000002 00578 .LONG 2
000004E8 000004DD 0057C P.AEJ: .LONG 1245, 1256
00 00 00 00 00584 P.AEN: .BYTE 0, 0
0000 00586 .WORD 1
00 00 00 00 00588 .BYTE 0, 0, 0, 0
00 2C 01 00 0058C .ASCII <1>\.\
00 00 00 00 0058E P.AEO: .BYTE 0, 0
0000 00590 .WORD 10
00 00 00 00 00592 .BYTE 0, 0, 0, 0
00 3E 01 00 00596 .ASCII <1>\>\
00000002 00598 .LONG 2
0000050B 00000501 0059C P.AEM: .LONG 1281, 1291
00 00 00 00 005A4 P.AEQ: .BYTE 0, 0
0000 005A6 .WORD 1
00 00 00 00 005AB .BYTE 0, 0, 0, 0
00 2C 01 00 005AC .ASCII <1>\.\
00 01 00 00 005AE P.AER: .BYTE 0, 1
0000 005B0 .WORD 2
00 00 00 00 005B2 .BYTE 0, 0, 0, 0
00 5D 01 00 005B6 .ASCII <1>\j\
00 00 00 00 005B8 P.AES: .BYTE 0, 0
0000 005BA .WORD 15
00 00 00 00 005BC .BYTE 0, 0, 0, 0
00 2E 2E 02 005C0 .ASCII <2>\..\
005C3 .BLKB 1
00000003 005C4 .LONG 3
00000535 0000052B 00000521 005C8 P.AEP: .LONG 1313, 1323, 1333
000003FD 000003ED 005D4 TERM_POINTER_TBL:
005DC .LONG 1005, 1021
005E4 .BYTE 0[8]
0000043D 00000425 00000411 005F0 .LONG 1041, 1061, 1085
005F4 .BYTE 0[4]
00000489 00000459 005FC .LONG 1113, 1161
005FC .BYTE 0[4]
000004F9 000004D9 000004C1 000004A5 00600 .LONG 1189, 1217, 1241, 1273
00610 .BLKB 4
```

00# 00614 BASE_CHARACTER_TABLE:							
						.BYTE	0[36]
						.LONG	131072
						.BYTE	0[88]
						.LONG	131072, 0, 1024, 0, 6, 0, 0, 1024, -
00000000	00000006	00000000	00000400	00000000	00020000	0063B	296960, 296960, 43008, 34880, 262144, -
00008840	0000A800	00048800	00048800	00000400	00000000	0063C	34896, 100408, 34816, 542, 542, 542, 542, -
0000021E	0000021E	00008800	00018938	00008850	00040000	006C4	542, 542, 542, 542, 542, 542, 542, 542, -
0000021E	0000021E	0000021E	0000021E	0000021E	0000021E	006DC	32768, 262144, 262144, 0, 34816, 295, -
00040000	00008000	00000000	00040000	0000021E	0000021E	006F4	262503, 295, 262519, 391, 295, 471, 263, -
00000127	00040167	00000127	00008800	00000000	00040000	0070C	263, 263, 263, 263, 263, 263, 263, -
00000107	00000107	00000107	00000127	00000187	00040177	00724	407, 263, 263, 262407, 263, 263, 262407, -
00000107	00000107	00000107	00000107	00000107	00000107	0073C	423, 263, 263, 0, 67584, 0, 65536, 6, 0, -
00000107	00040107	00000107	00000107	00000197	00000107	00754	295, 359, 295, 262519, 391, 295, 471, -
00000000	00000107	00000107	000001A7	00040107	00000107	0076C	263, 263, 263, 263, 263, 263, 263, -
00000127	00000000	00000006	00010000	00000000	00010800	00784	263, 407, 263, 263, 262407, 263, 263, -
00000107	00000127	00000187	00040177	00000127	00000167	0079C	262407, 423, 263, 263, 0, 0, 0, 0
00000107	00000107	00000107	00000107	00000107	00000107	007B4	
00000107	00000107	00000197	00000107	00000107	00000107	007CC	
00000107	000001A7	00040107	00000107	00000107	00040107	007E4	
	00000000	00000000	00000000	00000000	00000107	007FC	
						00810	
5E010800	3E042800	3D042800	3C00A800	2F00A800	00000008	00A14	.BLKB 516
				5D040000	26000800	00A18	.LONG 8
					5B000800	00A30	.LONG 637536256, 788572160, 1006675968, -
						P.AET:	1023682560, 1040459776, 1577125888, -
							1526728704, 1560543232
						00 03	P.AEV: .BYTE 3, 0
						000D	.WORD 15
00 00 00 00 00 00				4C 51	32 32	00A3B	.BYTE 50, 50, 0, 0, 0, 0, 0, 0
					45 03	00A3A	.ASCII <3>\EQL\
						00 03	P.AEW: .BYTE 3, 0
						000E	.WORD 14
00 00 00 00 00 00				51 45	32 32	00A44	.BYTE 50, 50, 0, 0, 0, 0, 0, 0
					4E 03	00A48	.ASCII <3>\NEQ\
						00 03	P.AEX: .BYTE 3, 0
						000F	.WORD 15
00 00 00 00 00 00				52 54	32 32	00A4A	.BYTE 50, 50, 0, 0, 0, 0, 0, 0
					47 03	00A4C	.ASCII <3>\GTR\
						00 03	P.AEY: .BYTE 3, 0
						0011	.WORD 17
00 00 00 00 00 00				51 45	32 32	00A6A	.BYTE 50, 50, 0, 0, 0, 0, 0, 0
					47 03	00A6C	.ASCII <3>\GEQ\
						00 03	P.AEZ: .BYTE 3, 0
						0013	.WORD 19
00 00 00 00 00 00				53 53	32 32	00A7A	.BYTE 50, 50, 0, 0, 0, 0, 0, 0
					4C 03	00A7C	.ASCII <3>\LSS\
						00 03	P.AFA: .BYTE 3, 0
						0015	.WORD 21
00 00 00 00 00 00				51 45	32 32	00A84	.BYTE 50, 50, 0, 0, 0, 0, 0, 0
					4C 03	00A88	.ASCII <3>\LEQ\
						00 02	P.AFB: .BYTE 2, 0
						0017	.WORD 23
00 00 00 00 00 00				54 4F	C8 28	00A9A	.BYTE 40, -56, 0, 0, 0, 0, 0, 0
					4E 03	00A9C	.ASCII <3>\NOT\
						00 03	P.AFC: .BYTE 3, 0
						0018	.WORD 24
00 00 00 00 00 00				44 4E	1E 1E	00AA4	.BYTE 30, 30, 0, 0, 0, 0, 0, 0
					41 03	00AA8	.ASCII <3>\AND\
						00AB4	

						00 03	00AB8	P.AFD:	.BYTE	3 0	
						0019	00ABA		.WORD	25	
00	00	00	00	00	00	14 14	00ABC		.BYTE	20, 20, 0, 0, 0, 0, 0, 0	
					52	4F 02	00AC4		.ASCII	<2>\OR\	
						00 03	00AC7	P.AFE:	.BYTE	3 0	
						001A	00AC9		.WORD	26	
00	00	00	00	00	00	0A 0A	00ACB		.BYTE	10, 10, 0, 0, 0, 0, 0, 0	
					52	4F 03	00AD3		.ASCII	<3>\XOR\	
						00 03	00AD7	P.AFF:	.BYTE	3 0	
						001B	00AD9		.WORD	27	
00	00	00	00	00	00	0A 0A	00ADB		.BYTE	10, 10, 0, 0, 0, 0, 0, 0	
					56	51 03	00AE3		.ASCII	<3>\EQV\	
							00AE7		.BLKB	1	
						0000000B	00AE8		.LONG	11	
00000A05	000009F5	000009E5	000009D5	000009C5	000009B5	00000A15	00AEC	P.AEU:	.LONG	2485, 2501, 2517, 2533, 2549, 2565, 2581, -	
	00000A54	00000A44	00000A35	00000A25	00000A15		00B04			2597, 2613, 2628, 2644	
						01 02	00B18	P.AFH:	.BYTE	2, 1	
						0002	00B1A		.WORD	2	
00	00	00	00	00	00	00 00	00B1C		.BYTE	0 0 0, 0, 0, 0, 0, 0	
						5C 01	00B24		.ASCII	<1><92>	
						01 03	00B26	P.AFI:	.BYTE	3, 1	
						0003	00B28		.WORD	3	
00	00	00	00	00	00	00 00	00B2A		.BYTE	0 0 0, 0, 0, 0, 0, 0	
						5C 01	00B32		.ASCII	<1><92>	
						01 03	00B34	P.AFJ:	.BYTE	3, 1	
						0005	00B36		.WORD	5	
00	00	00	00	00	00	00 00	00B38		.BYTE	0 0 0, 0, 0, 0, 0, 0	
						2E 01	00B40		.ASCII	<1>\.\	
						01 04	00B42	P.AFK:	.BYTE	4, 1	
						0004	00B44		.WORD	4	
00	00	00	00	00	00	00 00	00B46		.BYTE	0 0 0, 0, 0, 0, 0, 0	
						28 01	00B4E		.ASCII	<1>\{\	
						01 04	00B50	P.AFL:	.BYTE	4, 1	
						0004	00B52		.WORD	4	
00	00	00	00	00	00	00 00	00B54		.BYTE	0 0 0, 0, 0, 0, 0, 0	
						5B 01	00B5C		.ASCII	<1>\{\	
						01 04	00B5E	P.AFM:	.BYTE	4, 1	
						0007	00B60		.WORD	7	
00	00	00	00	00	00	00 00	00B62		.BYTE	0 0 0, 0, 0, 0, 0, 0	
						5E 01	00B6A		.ASCII	<1>\4\	
						00 03	00B6C	P.AFN:	.BYTE	3 0	
						0023	00B6E		.WORD	35	
00	00	00	00	00	00	3C 3C	00B70		.BYTE	60, 60, 0, 0, 0, 0, 0, 0	
					2F	2F 02	00B78		.ASCII	<2>\//\	
						00 03	00B7B	P.AFO:	.BYTE	3 0	
						0023	00B7D		.WORD	35	
00	00	00	00	00	00	3C 3C	00B7F		.BYTE	60, 60, 0, 0, 0, 0, 0, 0	
						26 01	00B87		.ASCII	<1>\8\	
						00 03	00B89	P.AFP:	.BYTE	3, 0	
						0006	00B8B		.WORD	6	
00	00	00	00	00	00	3C 3C	00B8D		.BYTE	60, 60, 0, 0, 0, 0, 0, 0	
						2B 01	00B95		.ASCII	<1>\+\	
						00 03	00B97	P.AFQ:	.BYTE	3, 0	
						0007	00B99		.WORD	7	
00	00	00	00	00	00	3C 3C	00B9B		.BYTE	60, 60, 0, 0, 0, 0, 0, 0	
						2D 01	00BA3		.ASCII	<1>\-\	
						00 02	00BA5	P.AFR:	.BYTE	2, 0	

00000ADB	00000ACD	00000ABF	00000AB1	00000AA3	00000A95	00C74	P.AFG:	.LONG	2709.	2723.	2737.	2751.	2765.	2779.	2793.	-
00000B30	00000B22	00000B14	00000B06	00000AF8	00000AE9	00C8C			2808.	2822.	2836.	2850.	2864.	2878.	2892.	-
00000B86	00000B77	00000B69	00000B5A	00000B4C	00000B3E	00CA4			2906.	2921.	2935.	2950.	2965.	2979.	2994.	-
00000BDD	00000BCF	00000BC0	00000BB2	00000BA3	00000B95	00CBC			3008.	3023.	3037.					
					01 00	00CD4	P.AGG:	.BYTE	0.	1						
					0002	00CD6		.WORD	2.							
			00 00	00 00	5D 01	00CD8		.BYTE	0.	0.	0.	0.				
					01 00	00CDC		.ASCII	<f>\;							
					0002	00CDE	P.AGH:	.BYTE	0.	1						
			00 00	00 00	29 01	00CE0		.WORD	2.							
					02 00	00CE2		.BYTE	0.	0.	0.	0.				
					0003	00CE6		.ASCII	<f>\;							
			00 00	00 00	3A 01	00CE8	P.AGI:	.BYTE	0.	2						
					0001	00CEA		.WORD	3.							
					00 00	00CEC		.BYTE	0.	0.	0.	0.				
					00 00	00CF0		.ASCII	<f>\;							
					0001	00CF2	P.AGJ:	.BYTE	0.	0						
			00 00	00 00	2C 01	00CF4		.WORD	1.							
					00000004	00CF6		.BYTE	0.	0.	0.	0.				
					00000C51	00CFA		.ASCII	<f>\;							
00000C6F	00000C65	00000C5B			00000004	00CFC		.LONG	4.							
					01	00D00	P.AGF:	.LONG	3153.	3163.	3173.	3183				
					02	00D10	P.AGK:	.BYTE	1							
					0005	00D11		.BYTE	2							
					03	00D12		.WORD	3							
					03	00D14		.BYTE	3							
					0003	00D15		.BYTE	3							
					00	00D16		.WORD	3							
					0A	00D18		.BYTE	0							
					0021	00D19		.BYTE	10							
					01	00D1A		.WORD	33							
					04	00D1C		.BYTE	1							
					0012	00D1D		.BYTE	4							
					00	00D1E		.WORD	18							
					0A	00D20		.BYTE	0							
					0021	00D21		.BYTE	10							
					01	00D22		.WORD	33							
					02	00D24		.BYTE	1							
					0005	00D25		.BYTE	2							
					03	00D26		.WORD	3							
					03	00D28		.BYTE	3							
					0012	00D29		.BYTE	3							
					02	00D2A		.WORD	18							
					01	00D2C		.BYTE	2							
					000C	00D2D		.BYTE	1							
					06	00D2E		.WORD	12							
					01	00D30		.BYTE	6							
					000C	00D31		.BYTE	1							
					07	00D32		.WORD	12							
					01	00D34		.BYTE	7							
					000C	00D35		.BYTE	1							
					08	00D36		.WORD	12							
					01	00D38		.BYTE	8							
					000C	00D39		.BYTE	1							
					00	00D3A		.WORD	12							
					09	00D3C		.BYTE	0							
						00D3D		.BYTE	9							

0021	00D3E	.WORD	33
01	00D40	.BYTE	1
01	00D41	.BYTE	1
000C	00D42	.WORD	12
02	00D44	.BYTE	2
01	00D45	.BYTE	1
000C	00D46	.WORD	12
06	00D48	.BYTE	6
01	00D49	.BYTE	1
000C	00D4A	.WORD	12
07	00D4C	.BYTE	7
01	00D4D	.BYTE	1
000C	00D4E	.WORD	12
08	00D50	.BYTE	8
01	00D51	.BYTE	1
000C	00D52	.WORD	12
00	00D54	.BYTE	0
09	00D55	.BYTE	9
0021	00D56	.WORD	33
01	00D58	.BYTE	1
04	00D59	.BYTE	4
0012	00D5A	.WORD	18
03	00D5C	.BYTE	3
08	00D5D	.BYTE	8
0021	00D5E	.WORD	33
08	00D60	.BYTE	8
05	00D61	.BYTE	5
0019	00D62	.WORD	25
07	00D64	.BYTE	7
06	00D65	.BYTE	6
0019	00D66	.WORD	25
0D	00D68	.BYTE	13
12	00D69	.BYTE	18
0019	00D6A	.WORD	25
09	00D6C	.BYTE	9
07	00D6D	.BYTE	7
0019	00D6E	.WORD	25
00	00D70	.BYTE	0
09	00D71	.BYTE	9
0021	00D72	.WORD	33
01	00D74	.BYTE	1
01	00D75	.BYTE	1
001F	00D76	.WORD	31
04	00D78	.BYTE	4
01	00D79	.BYTE	1
001D	00D7A	.WORD	29
05	00D7C	.BYTE	5
01	00D7D	.BYTE	1
001D	00D7E	.WORD	29
00	00D80	.BYTE	0
08	00D81	.BYTE	8
0021	00D82	.WORD	33
01	00D84	.BYTE	1
01	00D85	.BYTE	1
001F	00D86	.WORD	31
00	00D88	.BYTE	0
09	00D89	.BYTE	9

.....

0021	00D8A	.WORD	33
01	00D8C	.BYTE	1
01	00D8D	.BYTE	1
001F	00D8E	.WORD	31
00	00D90	.BYTE	0
09	00D91	.BYTE	9
0021	00D92	.WORD	33
00	00D94	.BYTE	0
0B	00D95	.BYTE	11
0021	00D96	.WORD	33
00000000	00D98	.LONG	0
	00D9C	P.AGL: .BLKB	0
00000000	00D9C	.LONG	0
	00DA0	P.AGM: .BLKB	0
02	00DA0	P.AGN: .BYTE	2
01	00DA1	.BYTE	1
0008	00DA2	.WORD	8
03	00DA4	.BYTE	3
03	00DA5	.BYTE	3
000A	00DA6	.WORD	10
09	00DA8	.BYTE	9
0F	00DA9	.BYTE	15
000A	00DAA	.WORD	10
05	00DAC	.BYTE	5
04	00DAD	.BYTE	4
0011	00DAE	.WORD	17
04	00DB0	.BYTE	4
07	00DB1	.BYTE	7
0016	00DB2	.WORD	22
07	00DB4	.BYTE	7
0A	00DB5	.BYTE	10
001B	00DB6	.WORD	27
01	00DB8	.BYTE	1
0C	00DB9	.BYTE	12
0020	00DBA	.WORD	32
00000000	00DBC	.LONG	0
01	00DC0	.BYTE	1
02	00DC1	.BYTE	2
0020	00DC2	.WORD	32
00000000	00DC4	.LONG	0
03	00DC8	.BYTE	3
0E	00DC9	.BYTE	14
000A	00DCA	.WORD	10
09	00DCC	.BYTE	9
0F	00DCD	.BYTE	15
000A	00DCE	.WORD	10
05	00DD0	.BYTE	5
10	00DD1	.BYTE	16
0011	00DD2	.WORD	17
04	00DD4	.BYTE	4
12	00DD5	.BYTE	18
0016	00DD6	.WORD	22
07	00DD8	.BYTE	7
15	00DD9	.BYTE	21
001B	00DDA	.WORD	27
01	00DDC	.BYTE	1
17	00DDD	.BYTE	23

.....

					0020	00DDE		.WORD	32	
					00000000	00DE0		.LONG	0	
					05	00DE4		.BYTE	5	
					19	00DE5		.BYTE	25	
					0011	00DE6		.WORD	17	
					04	00DE8		.BYTE	4	
					1C	00DE9		.BYTE	28	
					0016	00DEA		.WORD	22	
					07	00DEC		.BYTE	7	
					1F	00DED		.BYTE	31	
					001B	00DEE		.WORD	27	
					01	00DF0		.BYTE	1	
					21	00DF1		.BYTE	33	
					0020	00DF2		.WORD	32	
					00000000	00DF4		.LONG	0	
					05	00DF8		.BYTE	5	
					24	00DF9		.BYTE	36	
					0011	00DFA		.WORD	17	
					04	00DFC		.BYTE	4	
					26	00DFD		.BYTE	38	
					0016	00DFE		.WORD	22	
					07	00E00		.BYTE	7	
					28	00E01		.BYTE	40	
					001B	00E02		.WORD	27	
					01	00E04		.BYTE	1	
					2A	00E05		.BYTE	42	
					0020	00E06		.WORD	32	
					00000000	00E08		.LONG	0	
					05	00E0C		.BYTE	5	
					2C	00E0D		.BYTE	44	
					0011	00E0E		.WORD	17	
					04	00E10		.BYTE	4	
					2D	00E11		.BYTE	45	
					0016	00E12		.WORD	22	
					07	00E14		.BYTE	7	
					2E	00E15		.BYTE	46	
					001B	00E16		.WORD	27	
					01	00E18		.BYTE	1	
					2F	00E19		.BYTE	47	
					0020	00E1A		.WORD	32	
					00000000	00E1C		.LONG	0	
					00000000	00E20		.LONG	0	
00000C7D	00000D1D	00000C8D	00000BF1	00000A69	00000995	00E24	P.AGO:	.LONG	2453, 2665, 3057, 3213, 3357, 3197, 3353, -	
00000000	00000000	00000001	00000000	00000D1D	00000D19	00E3C			3357, 0, 1, 0, 0, 0	
					00000000	00E54				
					00000009	00E58		.LONG	9	
3C00A800	2F00A800	26000800	230000C0	5F0000B2	27000000	00E5C	P.AGP:	.LONG	654311424, 1593835698, 587202752, -	
			2E058830	3E042800	3D042800	00E74			637536256, 788572160, 1006675968, -	
									1023682560, 1040459776, 772114480	
					00 02	00E80	P.AGR:	.BYTE	2 0	
					0017	00E82		.WORD	23	
					00 00 00 00 00 00	00E84		.BYTE	60, -56, 0, 0, 0, 0, 0, 0	
					54 4F	00E8C		.ASCII	<3>\NOT\	
					00 02	00E90	P.AGS:	.BYTE	2 0	
					002B	00E92		.WORD	43	
					00 00 00 00 00 00	00E94		.BYTE	60, -56, 0, 0, 0, 0, 0, 0	
					53 42 41 03	00E9C		.ASCII	<3>\ABS\	

						00 03	00EA0	P.AGT:	.BYTE	3, 0	
						0024	00EA2		.WORD	36	
00	00			00	00	32 32	00EA4		.BYTE	50, 50, 0, 0, 0, 0, 0, 0	
				44	4F	4D 03	00EAC		.ASCII	<3>\MOD\	
						00 03	00EB0	P.AGU:	.BYTE	3, 0	
						0025	00EB2		.WORD	37	
00	00	00	00	00	00	32 32	00EB4		.BYTE	50, 50, 0, 0, 0, 0, 0, 0	
				4D	45	52 03	00EBC		.ASCII	<3>\REM\	
						00 03	00EC0	P.AGV:	.BYTE	3, 0	
						0018	00EC2		.WORD	24	
00	00	00	00	00	00	0A 0A	00EC4		.BYTE	10, 10, 0, 0, 0, 0, 0, 0	
				44	4E	41 03	00ECC		.ASCII	<3>\AND\	
						00 03	00ED0	P.AGW:	.BYTE	3, 0	
						0019	00ED2		.WORD	25	
00	00	00	00	00	00	0A 0A	00ED4		.BYTE	10, 10, 0, 0, 0, 0, 0, 0	
					52	4F 02	00EDC		.ASCII	<2>\OR\	
						00 03	00EDF	P.AGX:	.BYTE	3, 0	
						001A	00EE1		.WORD	26	
00	00	00	00	00	00	0A 0A	00EE3		.BYTE	10, 10, 0, 0, 0, 0, 0, 0	
				52	4F	58 03	00EEB		.ASCII	<3>\XOR\	
							00EEF		.BLKB	1	
						00000007	00EF0		.LONG	7	
00000E4D	00000E3D	00000E2D	00000E1D	00000E0D		00000DFD	00EF4	P.AGQ:	.LONG	3581, 3597, 3613, 3629, 3645, 3661, 3676	
						00000E5C	00F0C				
						01 02	00F10	P.AGZ:	.BYTE	2, 1	
						0002	00F12		.WORD	2	
00	00	00	00	00	00	00 00	00F14		.BYTE	0, 0, 0, 0, 0, 0, 0, 0	
						5C 01	00F1C		.ASCII	<1><92>	
						01 03	00F1E	P.AHA:	.BYTE	3, 1	
						0003	00F20		.WORD	3	
00	00	00	00	00	00	00 00	00F22		.BYTE	0, 0, 0, 0, 0, 0, 0, 0	
						5C 01	00F2A		.ASCII	<1><92>	
						01 03	00F2C	P.AHB:	.BYTE	3, 1	
						0005	00F2E		.WORD	5	
00	00	00	00	00	00	00 00	00F30		.BYTE	0, 0, 0, 0, 0, 0, 0, 0	
						2E 01	00F38		.ASCII	<1>\.\	
						01 04	00F3A	P.AHC:	.BYTE	4, 1	
						0004	00F3C		.WORD	4	
00	00	00	00	00	00	00 00	00F3E		.BYTE	0, 0, 0, 0, 0, 0, 0, 0	
						28 01	00F46		.ASCII	<1>\(\	
						02 02	00F48	P.AHD:	.BYTE	2, 2	
						000B	00F4A		.WORD	11	
00	00	00	00	00	00	C8 05	00F4C		.BYTE	5, -56, 0, 0, 0, 0, 0, 0	
						28 01	00F54		.ASCII	<1>\(\	
						02 04	00F56	P.AHE:	.BYTE	4, 2	
						000C	00F58		.WORD	12	
00	00	00	00	00	00	06 C8	00F5A		.BYTE	-56, 6, 0, 0, 0, 0, 0, 0	
						29 01	00F62		.ASCII	<1>\)\	
						00 03	00F64	P.AHF:	.BYTE	3, 0	
						000A	00F66		.WORD	10	
00	00	00	00	00	00	3C 3C	00F68		.BYTE	60, 60, 0, 0, 0, 0, 0, 0	
					2A	2A 02	00F70		.ASCII	<2>**\	
						00 03	00F73	P.AHG:	.BYTE	3, 0	
						0008	00F75		.WORD	8	
00	00	00	00	00	00	32 32	00F77		.BYTE	50, 50, 0, 0, 0, 0, 0, 0	
						2A 01	00F7F		.ASCII	<1>*\	
						00 03	00F81	P.AHH:	.BYTE	3, 0	

44	45	4E	49	41	52	54	53	4E	4F	43	0B	01090	P.AHU:	.ASCII	<1>\CONSTRAINED\	
				00	00	54	53	52	49	46	05	0109C	P.AHV:	.ASCII	<5>\FIRST\<0><0>	
				00	00	00	54	53	41	4C	04	010A4	P.AHW:	.ASCII	<4>\LAST\<0><0><0>	
				00	48	54	47	4E	45	4C	06	010AC	P.AHX:	.ASCII	<6>\LENGTH\<0>	
								53	4F	50	03	010B4	P.AHY:	.ASCII	<3>\POS\	
				00	00	00	44	45	52	50	04	010B8	P.AHZ:	.ASCII	<4>\PRED\<0><0><0>	
				00	00	00	45	5A	49	53	04	010C0	P.AIA:	.ASCII	<4>\SIZE\<0><0><0>	
				00	00	00	43	43	55	53	04	010C8	P.AIB:	.ASCII	<4>\SUCC\<0><0><0>	
								4C	41	56	03	010D0	P.AIC:	.ASCII	<3>\VAL\	
											00#	010D4	ADA_TICK	TABLE:		
00001035	00001031	00001029	00001021	00001019	0000100D	010D8								.BYTE	0[4]	
			0000104D	00001045	0000103D	010F0								.LONG	4109, 4121, 4129, 4137, 4145, 4149, 4157, -	
					00	00	010FC	P.AIE:	.BYTE	0, 0						
					0002	010FE			.WORD	2						
				00	00	00	00	01100		.BYTE	0, 0, 0, 0					
					29	01	01104		.ASCII	<1>\;\						
					02	00	01106	P.AIF:	.BYTE	0, 2						
					0003	01108			.WORD	3						
				00	00	00	00	0110A		.BYTE	0, 0, 0, 0					
					3A	01	0110E		.ASCII	<1>\:\						
					00	00	01110	P.AIG:	.BYTE	0, 0						
					0003	01112			.WORD	3						
				00	00	00	00	01114		.BYTE	0, 0, 0, 0					
					2E	02	01118		.ASCII	<2>\..\						
					00	00	0111B	P.AIH:	.BYTE	0, 0						
					0001	0111D			.WORD	1						
				00	00	00	00	0111F		.BYTE	0, 0, 0, 0					
					2C	01	01123		.ASCII	<1>\;\						
							01125		.BLKB	3						
					00000004	01128			.LONG	4						
00001098	0000108D	00001083			00001079	0112C	P.AID:	.LONG	4217, 4227, 4237, 4248							
					00000000	0113C			.LONG	0						
						01140	P.AII:	.BLKB	0							
					00000000	01140			.LONG	0						
					01	01144	P.AIJ:	.BLKB	0							
					01	01144	P.AIK:	.BYTE	1							
					01	01145			.BYTE	1						
					0002	01146			.WORD	2						
					00	01148			.BYTE	0						
					0A	01149			.BYTE	10						
					003E	0114A			.WORD	62						
					01	0114C			.BYTE	1						
					01	0114D			.BYTE	1						
					0002	0114E			.WORD	2						
					0B	01150			.BYTE	11						
					01	01151			.BYTE	1						
					0008	01152			.WORD	8						
					0C	01154			.BYTE	12						
					11	01155			.BYTE	17						
					001F	01156			.WORD	31						
					03	01158			.BYTE	3						
					03	01159			.BYTE	3						
					000A	0115A			.WORD	10						
					08	0115C			.BYTE	8						
					01	0115D			.BYTE	1						
					0014	0115E			.WORD	20						

00	01160	.BYTE	0
09	01161	.BYTE	9
003E	01162	.WORD	62
01	01164	.BYTE	1
01	01165	.BYTE	1
0002	01166	.WORD	2
00	01168	.BYTE	0
0A	01169	.BYTE	10
003E	0116A	.WORD	62
01	0116C	.BYTE	1
01	0116D	.BYTE	1
000A	0116E	.WORD	10
0B	01170	.BYTE	11
01	01171	.BYTE	1
0012	01172	.WORD	18
03	01174	.BYTE	3
0A	01175	.BYTE	10
003E	01176	.WORD	62
08	01178	.BYTE	8
05	01179	.BYTE	5
0014	0117A	.WORD	20
07	0117C	.BYTE	7
06	0117D	.BYTE	6
0014	0117E	.WORD	20
0D	01180	.BYTE	13
12	01181	.BYTE	18
0014	01182	.WORD	20
09	01184	.BYTE	9
07	01185	.BYTE	7
0014	01186	.WORD	20
00	01188	.BYTE	0
09	01189	.BYTE	9
003E	0118A	.WORD	62
01	0118C	.BYTE	1
01	0118D	.BYTE	1
000A	0118E	.WORD	10
00	01190	.BYTE	0
0A	01191	.BYTE	10
003E	01192	.WORD	62
01	01194	.BYTE	1
01	01195	.BYTE	1
001A	01196	.WORD	26
04	01198	.BYTE	4
01	01199	.BYTE	1
0018	0119A	.WORD	24
05	0119C	.BYTE	5
01	0119D	.BYTE	1
0018	0119E	.WORD	24
00	011A0	.BYTE	0
0A	011A1	.BYTE	10
003E	011A2	.WORD	62
01	011A4	.BYTE	1
01	011A5	.BYTE	1
001A	011A6	.WORD	26
00	011A8	.BYTE	0
0A	011A9	.BYTE	10
003E	011AA	.WORD	62

.....

01	011AC	.BYTE	1
01	011AD	.BYTE	1
001A	011AE	.WORD	26
0B	011B0	.BYTE	11
01	011B1	.BYTE	1
001D	011B2	.WORD	29
00	011B4	.BYTE	0
09	011B5	.BYTE	9
003E	011B6	.WORD	62
01	011B8	.BYTE	1
01	011B9	.BYTE	1
001A	011BA	.WORD	26
00	011BC	.BYTE	0
0A	011BD	.BYTE	10
003E	011BE	.WORD	62
01	011C0	.BYTE	1
01	011C1	.BYTE	1
0021	011C2	.WORD	33
00	011C4	.BYTE	0
0A	011C5	.BYTE	10
003E	011C6	.WORD	62
01	011C8	.BYTE	1
01	011C9	.BYTE	1
0021	011CA	.WORD	33
02	011CC	.BYTE	2
01	011CD	.BYTE	1
0021	011CE	.WORD	33
06	011D0	.BYTE	6
01	011D1	.BYTE	1
0021	011D2	.WORD	33
07	011D4	.BYTE	7
01	011D5	.BYTE	1
0021	011D6	.WORD	33
08	011D8	.BYTE	8
01	011D9	.BYTE	1
0021	011DA	.WORD	33
0B	011DC	.BYTE	11
01	011DD	.BYTE	1
002A	011DE	.WORD	42
03	011E0	.BYTE	3
03	011E1	.BYTE	3
0030	011E2	.WORD	48
0C	011E4	.BYTE	12
01	011E5	.BYTE	1
0014	011E6	.WORD	20
00	011E8	.BYTE	0
09	011E9	.BYTE	9
003E	011EA	.WORD	62
01	011EC	.BYTE	1
01	011ED	.BYTE	1
0021	011EE	.WORD	33
02	011F0	.BYTE	2
01	011F1	.BYTE	1
0021	011F2	.WORD	33
06	011F4	.BYTE	6
01	011F5	.BYTE	1
0021	011F6	.WORD	33

.....

07	011F8	.BYTE	7
01	011F9	.BYTE	1
0021	011FA	.WORD	33
08	011FC	.BYTE	8
01	011FD	.BYTE	1
0021	011FE	.WORD	33
00	01200	.BYTE	0
0A	01201	.BYTE	10
003E	01202	.WORD	62
01	01204	.BYTE	1
01	01205	.BYTE	1
0030	01206	.WORD	48
02	01208	.BYTE	2
01	01209	.BYTE	1
0030	0120A	.WORD	48
06	0120C	.BYTE	6
01	0120D	.BYTE	1
0030	0120E	.WORD	48
07	01210	.BYTE	7
01	01211	.BYTE	1
0030	01212	.WORD	48
08	01214	.BYTE	8
01	01215	.BYTE	1
0030	01216	.WORD	48
08	01218	.BYTE	11
01	01219	.BYTE	1
0038	0121A	.WORD	56
0C	0121C	.BYTE	12
05	0121D	.BYTE	5
0014	0121E	.WORD	20
00	01220	.BYTE	0
0A	01221	.BYTE	10
003E	01222	.WORD	62
01	01224	.BYTE	1
01	01225	.BYTE	1
0030	01226	.WORD	48
02	01228	.BYTE	2
01	01229	.BYTE	1
0030	0122A	.WORD	48
06	0122C	.BYTE	6
01	0122D	.BYTE	1
0030	0122E	.WORD	48
07	01230	.BYTE	7
01	01231	.BYTE	1
0030	01232	.WORD	48
08	01234	.BYTE	8
01	01235	.BYTE	1
0030	01236	.WORD	48
00	01238	.BYTE	0
0A	01239	.BYTE	10
003E	0123A	.WORD	62
00	0123C	.BYTE	0
08	0123D	.BYTE	11
003E	0123E	.WORD	62
02	01240	.BYTE	2
01	01241	.BYTE	1
0008	01242	.WORD	8

P.AIL:

.....

DI VI
00

03	01244	.BYTE	3
03	01245	.BYTE	3
000A	01246	.WORD	10
09	01248	.BYTE	9
0F	01249	.BYTE	15
000A	0124A	.WORD	10
05	0124C	.BYTE	5
04	0124D	.BYTE	4
0011	0124E	.WORD	17
04	01250	.BYTE	4
07	01251	.BYTE	7
0016	01252	.WORD	22
08	01254	.BYTE	11
30	01255	.BYTE	48
001A	01256	.WORD	26
01	01258	.BYTE	1
0C	01259	.BYTE	12
001A	0125A	.WORD	26
00000000	0125C	.LONG	0
01	01260	.BYTE	1
02	01261	.BYTE	2
001A	01262	.WORD	26
00000000	01264	.LONG	0
03	01268	.BYTE	3
0E	01269	.BYTE	14
000A	0126A	.WORD	10
09	0126C	.BYTE	9
0F	0126D	.BYTE	15
000A	0126E	.WORD	10
05	01270	.BYTE	5
10	01271	.BYTE	16
0011	01272	.WORD	17
04	01274	.BYTE	4
12	01275	.BYTE	18
0016	01276	.WORD	22
08	01278	.BYTE	11
31	01279	.BYTE	49
001A	0127A	.WORD	26
01	0127C	.BYTE	1
17	0127D	.BYTE	23
001A	0127E	.WORD	26
00000000	01280	.LONG	0
05	01284	.BYTE	5
19	01285	.BYTE	25
0011	01286	.WORD	17
04	01288	.BYTE	4
1C	01289	.BYTE	28
0016	0128A	.WORD	22
08	0128C	.BYTE	11
32	0128D	.BYTE	50
001A	0128E	.WORD	26
01	01290	.BYTE	1
21	01291	.BYTE	33
001A	01292	.WORD	26
00000000	01294	.LONG	0
05	01298	.BYTE	5
24	01299	.BYTE	36

.....

					0011	0129A	.WORD	17	
					04	0129C	.BYTE	4	
					26	0129D	.BYTE	38	
					0016	0129E	.WORD	22	
					01	012A0	.BYTE	1	
					2A	012A1	.BYTE	42	
					001A	012A2	.WORD	26	
					00000000	012A4	.LONG	0	
					00000000	012A8	.LONG	0	
000010A9	000011BD	000010C1	00000FAD	00000E71	00000DD9	012AC	P.AIM: .LONG	3545, 3697, 4013, 4289, 4541, 4265, 4285, -	
00000000	00000000	00000001	00000000	000010C1	000010BD	012C4		4289, 0, 1, 0, 0, 0	
					00000000	012DC			
					00000007	012E0	P.AIN: .LONG	7	
3E042800	3C04A800	5E010800	3A042800	25000004	2E01803E	012E4		771850302, 620756996, 973350912, -	
					3D042800	012FC		1577125888, 1006938112, 1040459776, -	
								1023682560	
					00 02	01300	P.AIP: .BYTE	2, 0	
					001E	01302	.WORD	30	
	00 00 00 00	00 00	00 00	00 00	C8 2D	01304	.BYTE	45, -56, 0, 0, 0, 0, 0, 0	
				54 4F	4E 03	0130C	.ASCII	<3>\NOT\	
					00 03	01310	P.AIQ: .BYTE	3, 0	
					001F	01312	.WORD	31	
	00 00 00 00	00 00	00 00	00 00	28 28	01314	.BYTE	40, 40, 0, 0, 0, 0, 0, 0	
				44 4E	41 03	0131C	.ASCII	<3>\AND\	
					00 03	01320	P.AIR: .BYTE	3, 0	
					0020	01322	.WORD	32	
	00 00 00 00	00 00	00 00	00 00	1E 1E	01324	.BYTE	30, 30, 0, 0, 0, 0, 0, 0	
				52	4F 02	0132C	.ASCII	<2>\OR\	
					00 03	0132F	P.AIS: .BYTE	3, 0	
					0021	01331	.WORD	33	
	00 00 00 00	00 00	00 00	00 00	1E 1E	01333	.BYTE	30, 30, 0, 0, 0, 0, 0, 0	
				52 4F	58 03	0133B	.ASCII	<3>\XOR\	
					00 03	0133F	P.AIT: .BYTE	3, 0	
					003A	01341	.WORD	58	
	00 00 00 00	00 00	00 00	00 00	14 14	01343	.BYTE	20, 20, 0, 0, 0, 0, 0, 0	
				50 4D	49 03	0134B	.ASCII	<3>\IMP\	
					00 03	0134F	P.AIU: .BYTE	3, 0	
					0022	01351	.WORD	34	
	00 00 00 00	00 00	00 00	00 00	0A 0A	01353	.BYTE	10, 10, 0, 0, 0, 0, 0, 0	
				56 51	45 03	0135B	.ASCII	<3>\EQV\	
						0135F	.BLKB	1	
					00000006	01360	.LONG	6	
000012CC	000012BC	000012AC	0000129D	0000128D	0000127D	01364	P.AIO: .LONG	4733, 4749, 4765, 4780, 4796, 4812	
					01 02	0137C	P.AIW: .BYTE	2, 1	
					0002	0137E	.WORD	2	
	00 00 00 00	00 00	00 00	00 00	00 00	01380	.BYTE	0, 0, 0, 0, 0, 0, 0, 0	
					5C 01	01388	.ASCII	<1><2>	
					01 03	0138A	P.AIX: .BYTE	3, 1	
					0003	0138C	.WORD	3	
	00 00 00 00	00 00	00 00	00 00	00 00	0138E	.BYTE	0, 0, 0, 0, 0, 0, 0, 0	
					5C 01	01396	.ASCII	<1><2>	
					01 03	01398	P.AIY: .BYTE	3, 1	
					0005	0139A	.WORD	5	
	00 00 00 00	00 00	00 00	00 00	00 00	0139C	.BYTE	0, 0, 0, 0, 0, 0, 0, 0	
				3A	3A 02	013A4	.ASCII	<2>\::\	
					01 04	013A7	P.AIZ: .BYTE	4, 1	
					0004	013A9	.WORD	4	

00	00	00	00	00	00	00	00	013AB	.BYTE	0	0	0	0	0	0	0
						28	01	013B3	.ASCII	<1>\{\						
						02	02	013B5	P.AJA:	.BYTE	2	2				
							000B	013B7	.WORD	11						
00	00	00	00	00	00	C8	05	013B9	.BYTE	5	-56	0	0	0	0	0
						28	01	013C1	.ASCII	<1>\{\						
						02	04	013C3	P.AJB:	.BYTE	4	2				
							000C	013C5	.WORD	12						
00	00	00	00	00	00	06	C8	013C7	.BYTE	-56	6	0	0	0	0	0
						29	01	013CF	.ASCII	<1>\)\						
						00	02	013D1	P.AJC:	.BYTE	2	0				
							0004	013D3	.WORD	4						
00	00	00	00	00	00	C8	46	013D5	.BYTE	70	-56	0	0	0	0	0
						2B	01	013DD	.ASCII	<1>\+\						
						00	02	013DF	P.AJD:	.BYTE	2	0				
							0005	013E1	.WORD	5						
00	00	00	00	00	00	C8	46	013E3	.BYTE	70	-56	0	0	0	0	0
						2D	01	013EB	.ASCII	<1>\-\						
						00	03	013ED	P.AJE:	.BYTE	3	0				
							000A	013EF	.WORD	10						
00	00	00	00	00	00	5C	5A	013F1	.BYTE	90	92	0	0	0	0	0
					2A	2A	02	013F9	.ASCII	<2>**\						
						00	03	013FC	P.AJF:	.BYTE	3	0				
							000A	013FE	.WORD	10						
00	00	00	00	00	00	5C	5A	01400	.BYTE	90	92	0	0	0	0	0
						5E	01	01408	.ASCII	<1>\^\						
						00	03	0140A	P.AJG:	.BYTE	3	0				
							0008	0140C	.WORD	8						
00	00	00	00	00	00	50	50	0140E	.BYTE	80	80	0	0	0	0	0
						2A	01	01416	.ASCII	<1>*\						
						00	03	01418	P.AJH:	.BYTE	3	0				
							0009	0141A	.WORD	9						
00	00	00	00	00	00	50	50	0141C	.BYTE	80	80	0	0	0	0	0
						2F	01	01424	.ASCII	<1>\/\						
						00	03	01426	P.AJI:	.BYTE	3	0				
							0006	01428	.WORD	6						
00	00	00	00	00	00	3C	3C	0142A	.BYTE	60	60	0	0	0	0	0
						2B	01	01432	.ASCII	<1>\+\						
						00	03	01434	P.AJJ:	.BYTE	3	0				
							0007	01436	.WORD	7						
00	00	00	00	00	00	3C	3C	01438	.BYTE	60	60	0	0	0	0	0

					000F	0156E		.WORD	15	
					04	01570		.BYTE	4	
					07	01571		.BYTE	7	
					0013	01572		.WORD	19	
					01	01574		.BYTE	1	
					0C	01575		.BYTE	12	
					0017	01576		.WORD	23	
					00000000	01578		.LONG	0	
					01	0157C		.BYTE	1	
					02	0157D		.BYTE	2	
					0017	0157E		.WORD	23	
					00000000	01580		.LONG	0	
					03	01584		.BYTE	3	
					0E	01585		.BYTE	14	
					0009	01586		.WORD	9	
					09	01588		.BYTE	9	
					0F	01589		.BYTE	15	
					0009	0158A		.WORD	9	
					05	0158C		.BYTE	5	
					10	0158D		.BYTE	16	
					000F	0158E		.WORD	15	
					04	01590		.BYTE	4	
					12	01591		.BYTE	18	
					0013	01592		.WORD	19	
					01	01594		.BYTE	1	
					17	01595		.BYTE	23	
					0017	01596		.WORD	23	
					00000000	01598		.LONG	0	
					05	0159C		.BYTE	5	
					19	0159D		.BYTE	25	
					000F	0159E		.WORD	15	
					04	015A0		.BYTE	4	
					1C	015A1		.BYTE	28	
					0013	015A2		.WORD	19	
					01	015A4		.BYTE	1	
					21	015A5		.BYTE	33	
					0017	015A6		.WORD	23	
					00000000	015A8		.LONG	0	
					05	015AC		.BYTE	5	
					24	015AD		.BYTE	36	
					000F	015AE		.WORD	15	
					04	015B0		.BYTE	4	
					26	015B1		.BYTE	38	
					0013	015B2		.WORD	19	
					01	015B4		.BYTE	1	
					2A	015B5		.BYTE	42	
					0017	015B6		.WORD	23	
					00000000	015B8		.LONG	0	
					00000000	015BC		.LONG	0	
000014C9	000014DD	00000C8D	00001449	000012E1	00001261	015C0	P.AKA:	.LONG	4705, 4833, 5193, 3213, 5341, 5321, 5337, -	
00000000	00000000	00000001	00000000	000014DD	000014D9	015D8			5341, 0, 1, 0, 0, 1	
					00000001	015F0				
					00000007	015F4		.LONG	7	
5E010800	5D040000	5B000800	3E040000	3C008800	24000007	015F8	P.AKB:	.LONG	603979783, 1006667776, 1040449536, -	
					5F000007	01610			1526728704, 1560543232, 1577125888, -	
									1593835527	
					00 03	01614	P.AKD:	.BYTE	3, 0	

00	00	00	00	00	00	32	32	01700	.BYTE	50, 50, 0, 0, 0, 0, 0, 0	
			55	53	53	4C	04	01708	.ASCII	<4>\LSSU\	
						00	03	0170D	P.AKS: .BYTE	3, 0	
						00	14	0170F	.WORD	20	
00	00	00	00	00	00	32	32	01711	.BYTE	50, 50, 0, 0, 0, 0, 0, 0	
			41	53	53	4C	04	01719	.ASCII	<4>\LSSA\	
						00	03	0171E	P.AKT: .BYTE	3, 0	
						00	15	01720	.WORD	21	
00	00	00	00	00	00	32	32	01722	.BYTE	50, 50, 0, 0, 0, 0, 0, 0	
				51	45	4C	03	0172A	.ASCII	<3>\LEQ\	
						00	03	0172E	P.AKU: .BYTE	3, 0	
						00	16	01730	.WORD	22	
00	00	00	00	00	00	32	32	01732	.BYTE	50, 50, 0, 0, 0, 0, 0, 0	
			55	51	45	4C	04	0173A	.ASCII	<4>\LEQU\	
						00	03	0173F	P.AKV: .BYTE	3, 0	
						00	16	01741	.WORD	22	
00	00	00	00	00	00	32	32	01743	.BYTE	50, 50, 0, 0, 0, 0, 0, 0	
			41	51	45	4C	04	0174B	.ASCII	<4>\LEQA\	
						00	02	01750	P.AKW: .BYTE	2, 0	
						00	1E	01752	.WORD	30	
00	00	00	00	00	00	C8	28	01754	.BYTE	40, -56, 0, 0, 0, 0, 0, 0	
				54	4F	4E	03	0175C	.ASCII	<3>\NOT\	
						00	03	01760	P.AKX: .BYTE	3, 0	
						00	1F	01762	.WORD	31	
00	00	00	00	00	00	1E	1E	01764	.BYTE	30, 30, 0, 0, 0, 0, 0, 0	
				44	4E	41	03	0176C	.ASCII	<3>\AND\	
						00	03	01770	P.AKY: .BYTE	3, 0	
						00	20	01772	.WORD	32	
00	00	00	00	00	00	14	14	01774	.BYTE	20, 20, 0, 0, 0, 0, 0, 0	
					52	4F	02	0177C	.ASCII	<2>\OR\	
						00	03	0177F	P.AKZ: .BYTE	3, 0	
						00	22	01781	.WORD	34	
00	00	00	00	00	00	0A	0A	01783	.BYTE	10, 10, 0, 0, 0, 0, 0, 0	
				56	51	45	03	0178B	.ASCII	<3>\EQV\	
						00	03	0178F	P.ALA: .BYTE	3, 0	
						00	21	01791	.WORD	33	
00	00	00	00	00	00	0A	0A	01793	.BYTE	10, 10, 0, 0, 0, 0, 0, 0	
				52	4F	58	03	0179B	.ASCII	<3>\XOR\	
								0179F	.BLKB	1	
								017A0	.LONG	24	
000015E3	000015D3	000015C2	000015B1	000015A1	00000018	00001591	017A4	P.AKC: .LONG	5521, 5537, 5553, 5570, 5587, 5603, 5620, -		
00001647	00001637	00001626	00001615	00001605	000015F4	017BC			5637, 5653, 5670, 5687, 5703, 5720, 5737, -		
000016AB	0000169B	0000168A	00001679	00001669	00001658	017D4			5753, 5770, 5787, 5803, 5820, 5837, 5853, -		
0000170C	000016FC	000016ED	000016DD	000016CD	000016BC	017EC			5869, 5884, 5900		
					01	02	01804	P.ALC: .BYTE	2, 1		
					00	02	01806	.WORD	2		
00	00	00	00	00	00	00	00	01808	.BYTE	0, 0, 0, 0, 0, 0, 0, 0	
					5C	01	01810	.ASCII	<1><92>		
					01	03	01812	P.ALD: .BYTE	3, 1		
					00	03	01814	.WORD	3		
00	00	00	00	00	00	00	00	01816	.BYTE	0, 0, 0, 0, 0, 0, 0, 0	
					5C	01	0181E	.ASCII	<1><92>		
					01	04	01820	P.ALE: .BYTE	4, 1		
					00	04	01822	.WORD	4		
00	00	00	00	00	00	00	00	01824	.BYTE	0, 0, 0, 0, 0, 0, 0, 0	
					58	01	0182C	.ASCII	<1>\[\		
					02	02	0182E	P.ALF: .BYTE	2, 2		

00	00	00	00	00	00	0008	01830	.WORD	11	
						C8 05	01832	.BYTE	5	-56, 0, 0, 0, 0, 0, 0
						28 01	0183A	.ASCII	<1>\(\	
						02 04	0183C	P.ALG: .BYTE	4	2
						000C	0183E	.WORD	12	
00	00	00	00	00	00	06 C8	01840	.BYTE	-56	6, 0, 0, 0, 0, 0, 0
						29 01	01848	.ASCII	<1>\(\	
						02 04	0184A	P.ALH: .BYTE	4	2
						0031	0184C	.WORD	46	
00	00	00	00	00	00	6E C8	0184E	.BYTE	-56	110, 0, 0, 0, 0, 0, 0
						3C 01	01856	.ASCII	<1>\(<\	
						00 02	01858	P.ALI: .BYTE	2	0
						0003	0185A	.WORD	3	
00	00	00	00	00	00	C8 64	0185C	.BYTE	100	-56, 0, 0, 0, 0, 0, 0
						2E 01	01864	.ASCII	<1>\(\	
						00 02	01866	P.ALJ: .BYTE	2	0
						0004	01868	.WORD	4	
00	00	00	00	00	00	C8 5A	0186A	.BYTE	90	-56, 0, 0, 0, 0, 0, 0
						2B 01	01872	.ASCII	<1>\(+\	
						00 02	01874	P.ALK: .BYTE	2	0
						0005	01876	.WORD	5	
00	00	00	00	00	00	C8 5A	01878	.BYTE	90	-56, 0, 0, 0, 0, 0, 0
						2D 01	01880	.ASCII	<1>\(-\	
						00 03	01882	P.ALL: .BYTE	3	0
						0026	01884	.WORD	38	
00	00	00	00	00	00	50 50	01886	.BYTE	80	80, 0, 0, 0, 0, 0, 0
						5E 01	0188E	.ASCII	<1>\^-\	
						00 03	01890	P.ALM: .BYTE	3	0
						0008	01892	.WORD	8	
00	00	00	00	00	00	46 46	01894	.BYTE	70	70, 0, 0, 0, 0, 0, 0
						2A 01	0189C	.ASCII	<1>*\	
						00 03	0189E	P.ALN: .BYTE	3	0
						0009	018A0	.WORD	9	
00	00	00	00	00	00	46 46	018A2	.BYTE	70	70, 0, 0, 0, 0, 0, 0
						2F 01	018AA	.ASCII	<1>\/\	
						00 03	018AC	P.ALO: .BYTE	3	0
						0006	018AE	.WORD	6	
00	00	00	00	00	00	3C 3C	018B0	.BYTE	60	60, 0, 0, 0, 0, 0, 0
						2B 01	018B8	.ASCII	<1>\(+\	
						00 03	018BA	P. ALP: .BYTE	3	0
						0007	018BC	.WORD	7	
00	00	00	00	00	00	3C 3C	018BE	.BYTE	60	60, 0, 0, 0, 0, 0, 0
						2D 01	018C6	.ASCII	<1>\(-\	
						0000000E	018C8	.LONG	14	
000017C7	000017B9	000017AB	0000179D	0000178F	00001781	00001781	018CC	P. ALB: .LONG	6017, 6031, 6045, 6059, 6073, 6087, 6101, -	
0000181B	0000180D	000017FF	000017F1	000017E3	000017D5	000017D5	018E4		6115, 6129, 6143, 6157, 6171, 6185, 6199	
				00001837	00001829	00001829	018FC			
					00 00	00 00	01904	P. ALR: .BYTE	0	0
					0002	00 00	01906	.WORD	2	
				00 00	00 00	00 00	01908	.BYTE	0	0, 0, 0
				5D 01	00 00	00 00	0190C	.ASCII	<1>\j\	
				02 00	00 00	00 00	0190E	P. ALS: .BYTE	0	2
				0003	00 00	00 00	01910	.WORD	3	
				00 00	00 00	00 00	01912	.BYTE	0	0, 0, 0
				3A 01	00 00	00 00	01916	.ASCII	<1>\:\	
				00 00	00 00	00 00	01918	P. ALT: .BYTE	0	0
				0001	00 00	00 00	0191A	.WORD	1	


```
00 00 00 00 0191C .BYTE 0 0, 0, 0
          2C 01 01920 .ASCII <f>\,\
          01922 .BLKB 2
00000003 01924 .LONG 3
00001895 00001888 00001881 01928 P.ALQ: .LONG 6273, 6283, 6293
00000000 01934 .LONG 0
          00000000 01938 P.ALU: .BLKB 0
          01938 .LONG 0
          02 0193C P.ALV: .BLKB 0
          01 0193C P.ALW: .BYTE 2
          0006 0193D .BYTE 1
          03 0193E .WORD 6
          03 01940 .BYTE 3
          03 01941 .BYTE 3
0008 01942 .WORD 8
          09 01944 .BYTE 9
          0F 01945 .BYTE 15
0008 01946 .WORD 8
          04 01948 .BYTE 4
          08 01949 .BYTE 8
000D 0194A .WORD 13
          01 0194C .BYTE 1
          0C 0194D .BYTE 12
000F 0194E .WORD 15
00000000 01950 .LONG 0
          01 01954 .BYTE 1
          02 01955 .BYTE 2
000F 01956 .WORD 15
00000000 01958 .LONG 0
          03 0195C .BYTE 3
          0E 0195D .BYTE 14
0008 0195E .WORD 8
          09 01960 .BYTE 9
          0F 01961 .BYTE 15
0008 01962 .WORD 8
          04 01964 .BYTE 4
          14 01965 .BYTE 20
000D 01966 .WORD 13
          01 01968 .BYTE 1
          17 01969 .BYTE 23
000F 0196A .WORD 15
00000000 0196C .LONG 0
          01 01970 .BYTE 1
          2A 01971 .BYTE 42
000F 01972 .WORD 15
00000000 01974 .LONG 0
00000000 01978 .LONG 0
000018A5 000018B9 00000C8D 00001849 00001721 00001575 0197C P.ALX: .LONG 5493, 5921, 6217, 3213, 6329, 6309, 6325, -
00000000 00000000 00000001 00000000 000018B9 000018B5 01994 .LONG 6329, 0, 1, 0, 0, 0
00000000 019AC .LONG 13
00000000 01980 P.ALY: .LONG 603979783, 1593835527, 553658368, -
3C00A800 25000800 2A008800 21002800 5F000007 24000007 01984 .LONG 704677888, 620759040, 1006675968, -
7C002800 5E010800 5D040000 5B000800 3E042800 3D042800 019CC .LONG 1023682560, 1040459776, 1526728704, -
7E000800 019E4 .LONG 1560543232, 1577125888, 2080385024, -
          2113931264
00 02 019E8 P.AMA: .BYTE 2, 0
```

00	00	00	00	00	00	00	00	0029	019EA	.WORD	41	
	46	4F	45	5A	49	53	06	019EC	.BYTE	-116	-56	0. 0. 0. 0. 0. 0
								019F4	.ASCII	<6>\\$	SIZEOF\	
								019FB	.BLKB	1		
						00000001		019FC	.LONG	1		
						00001965		01A00	P.ALZ:	.LONG	6501	
						01	02	01A04	P.AMC:	.BYTE	2, 1	
						0002		01A06	.WORD	2		
00	00	00	00	00	00	00	00	01A08	.BYTE	0	0	0. 0. 0. 0. 0. 0
						5C	01	01A10	.ASCII	<1><62>		
						01	03	01A12	P.AMD:	.BYTE	3, 1	
						0003		01A14	.WORD	3		
00	00	00	00	00	00	00	00	01A16	.BYTE	0	0	0. 0. 0. 0. 0. 0
						5C	01	01A1E	.ASCII	<1><62>		
						01	04	01A20	P.AME:	.BYTE	4, 1	
						0004		01A22	.WORD	4		
00	00	00	00	00	00	00	00	01A24	.BYTE	0	0	0. 0. 0. 0. 0. 0
						5B	01	01A2C	.ASCII	<1>\[
						01	03	01A2E	P.AMF:	.BYTE	3, 1	
						0005		01A30	.WORD	5		
00	00	00	00	00	00	00	00	01A32	.BYTE	0	0	0. 0. 0. 0. 0. 0
						2E	01	01A3A	.ASCII	<1>\.		
						02	02	01A3C	P.AMG:	.BYTE	2, 2	
						000B		01A3E	.WORD	11		
00	00	00	00	00	00	C8	05	01A40	.BYTE	5	-56	0. 0. 0. 0. 0. 0
						28	01	01A48	.ASCII	<1>\(
						02	04	01A4A	P.AMH:	.BYTE	4, 2	
						000C		01A4C	.WORD	12		
00	00	00	00	00	00	06	C8	01A4E	.BYTE	-56	6	0. 0. 0. 0. 0. 0
						29	01	01A56	.ASCII	<1>\)		
						00	02	01A58	P.AMI:	.BYTE	2, 0	
						0017		01A5A	.WORD	23		
00	00	00	00	00	00	C8	8C	01A5C	.BYTE	-116	-56	0. 0. 0. 0. 0. 0
						21	01	01A64	.ASCII	<1>\!		
						00	02	01A66	P.AMJ:	.BYTE	2, 0	
						001E		01A68	.WORD	30		
00	00	00	00	00	00	C8	8C	01A6A	.BYTE	-116	-56	0. 0. 0. 0. 0. 0
						7E	01	01A72	.ASCII	<1>\2		
						00	02	01A74	P.AMK:	.BYTE	2, 0	
						0003		01A76	.WORD	3		
00	00	00	00	00	00	C8	8C	01A78	.BYTE	-116	-56	0. 0. 0. 0. 0. 0
						2A	01	01A80	.ASCII	<1>*		
						00	03	01A82	P.AML:	.BYTE	3, 0	
						0008		01A84	.WORD	8		
00	00	00	00	00	00	82	82	01A86	.BYTE	-126	-126	0. 0. 0. 0. 0. 0
						2A	01	01A8E	.ASCII	<1>*		
						00	03	01A90	P.AMM:	.BYTE	3, 0	
						0009		01A92	.WORD	9		
00	00	00	00	00	00	82	82	01A94	.BYTE	-126	-126	0. 0. 0. 0. 0. 0
						2F	01	01A9C	.ASCII	<1>\/		
						00	03	01A9E	P.AMN:	.BYTE	3, 0	
						0025		01AA0	.WORD	37		
00	00	00	00	00	00	82	82	01AA2	.BYTE	-126	-126	0. 0. 0. 0. 0. 0
						25	01	01AAA	.ASCII	<1>\4		
						00	03	01AAC	P.AMO:	.BYTE	3, 0	
						0026		01AAE	.WORD	38		
00	00	00	00	00	00	6E	6E	01AB0	.BYTE	110	110	0. 0. 0. 0. 0. 0

```

3C 3C 02 01AB8 .ASCII <2>\<<\
00 03 01ABB P.AMP: .BYTE 3 0
0027 01ABD .WORD 3 0
00 00 00 00 00 00 6E 6E 01ABF .BYTE 110, 110, 0, 0, 0, 0, 0, 0
3E 02 01AC7 .ASCII <2>\>>\
00 03 01ACA P.AMQ: .BYTE 3 0
0013 01ACC .WORD 1 0
00 00 00 00 00 00 64 64 01ACE .BYTE 100, 100, 0, 0, 0, 0, 0, 0
3C 01 01AD6 .ASCII <1>\<\
00 03 01AD8 P.AMR: .BYTE 3 0
0015 01ADA .WORD 2 1
00 00 00 00 00 00 64 64 01ADC .BYTE 100, 100, 0, 0, 0, 0, 0, 0
3D 02 01AE4 .ASCII <2>\<=\
00 03 01AE7 P.AMS: .BYTE 3 0
000F 01AE9 .WORD 1 0
00 00 00 00 00 00 64 64 01AEB .BYTE 100, 100, 0, 0, 0, 0, 0, 0
3E 01 01AF3 .ASCII <1>\>\
00 03 01AF5 P.AMT: .BYTE 3 0
0011 01AF7 .WORD 1 0
00 00 00 00 00 00 64 64 01AF9 .BYTE 100, 100, 0, 0, 0, 0, 0, 0
3D 02 01B01 .ASCII <2>\>=\
00 03 01B04 P.AMU: .BYTE 3 0
000D 01B06 .WORD 1 0
00 00 00 00 00 00 5A 5A 01B08 .BYTE 90, 90, 0, 0, 0, 0, 0, 0
3D 02 01B10 .ASCII <2>\==\
00 03 01B13 P.AMV: .BYTE 3 0
000E 01B15 .WORD 1 0
00 00 00 00 00 00 5A 5A 01B17 .BYTE 90, 90, 0, 0, 0, 0, 0, 0
3D 02 01B1F .ASCII <2>\!=\
00 03 01B22 P.AMW: .BYTE 3 0
0021 01B24 .WORD 3 0
00 00 00 00 00 00 46 46 01B26 .BYTE 70, 70, 0, 0, 0, 0, 0, 0
5E 01 01B2E .ASCII <1>\*\
00 03 01B30 P.AMX: .BYTE 3 0
0020 01B32 .WORD 3 0
00 00 00 00 00 00 3C 3C 01B34 .BYTE 60, 60, 0, 0, 0, 0, 0, 0
7C 01 01B3C .ASCII <1>\!\
00 03 01B3E P.AMY: .BYTE 3 0
001D 01B40 .WORD 2 0
00 00 00 00 00 00 28 28 01B42 .BYTE 40, 40, 0, 0, 0, 0, 0, 0
7C 02 01B4A .ASCII <2>\!\!\
01B4D .BLKB 3
00000017 01B50 .LONG 23
00001981 01B54 P.AMB: .LONG 6529, 6543, 6557, 6571, 6585, 6599, 6613, -
00001981 01B6C 6627, 6641, 6655, 6669, 6683, 6697, 6712, -
000019D5 01B84 6727, 6741, 6756, 6770, 6785, 6800, 6815, -
00001A29 01B9C 6829, 6843
00 02 01BB0 P.AMZ: .BYTE 2 0
0028 01BB2 .WORD 4 0
00 00 00 00 00 00 C8 8C 01BB4 .BYTE -116, -56, 0, 0, 0, 0, 0, 0
26 01 01BBC .ASCII <1>\&\
00 03 01BBE P.ANA: .BYTE 3 0
001F 01BC0 .WORD 3 1
00 00 00 00 00 00 50 50 01BC2 .BYTE 80, 80, 0, 0, 0, 0, 0, 0
26 01 01BCA .ASCII <1>\&\
00 03 01BCC P.ANB: .BYTE 3 0
001C 01BCE .WORD 2 0

```

```

000019C7 000019B9 000019AB 0000199D 0000198F 00001981
00001A1B 00001A0D 000019FF 000019F1 000019E3 000019D5
00001A72 00001A64 00001A55 00001A47 00001A38 00001A29
00001ABB 00001AAD 00001A9F 00001A90

```

00	00	00	00	00	00	32	32	01B00	.BYTE	50, 50, 0, 0, 0, 0, 0, 0
					26	26	02	01B08	.ASCII	<2>\&&\
					00	03	01B0B	P.ANC: .BYTE	3, 0	
						0006	01B0D	.WORD	6	
00	00	00	00	00	00	78	78	01B0F	.BYTE	120, 120, 0, 0, 0, 0, 0, 0
						28	01	01BE7	.ASCII	<1>\+\
						00	02	01BE9	P.AND: .BYTE	2, 0
						0005	01BEB	.WORD	5	
00	00	00	00	00	00	C8	8C	01BED	.BYTE	-116, -56, 0, 0, 0, 0, 0, 0
						2D	01	01BF5	.ASCII	<1>\-\
						00	03	01BF7	P.ANE: .BYTE	3, 0
						0007	01BF9	.WORD	7	
00	00	00	00	00	00	78	78	01BF8	.BYTE	120, 120, 0, 0, 0, 0, 0, 0
						2D	01	01C03	.ASCII	<1>\-\
						01	03	01C05	P.ANF: .BYTE	3, 1
						0008	01C07	.WORD	8	
00	00	00	00	00	00	00	00	01C09	.BYTE	0, 0, 0, 0, 0, 0, 0, 0
					3E	2D	02	01C11	.ASCII	<2>\-\>\
						00	02	01C14	P.ANG: .BYTE	2, 0
						0003	01C16	.WORD	3	
00	00	00	00	00	00	C8	28	01C18	.BYTE	40, -56, 0, 0, 0, 0, 0, 0
						2A	01	01C20	.ASCII	<1>*\
						00	00	01C22	P.ANI: .BYTE	0, 0
						0002	01C24	.WORD	2	
				00	00	00	00	01C26	.BYTE	0, 0, 0, 0
						5D	01	01C2A	.ASCII	<1>\j\
						02	00	01C2C	P.ANJ: .BYTE	0, 2
						0003	01C2E	.WORD	3	
				00	00	00	00	01C30	.BYTE	0, 0, 0, 0
						3A	01	01C34	.ASCII	<1>\:\
						00	00	01C36	P.ANK: .BYTE	0, 0
						0001	01C38	.WORD	1	
				00	00	00	00	01C3A	.BYTE	0, 0, 0, 0
						2C	01	01C3E	.ASCII	<1>\,\
						00000003	01C40	.LONG	3	
00001BB3	00001BA9					00001B9F	01C44	P.ANH: .LONG	7071, 7081, 7091	
						00000000	01C50	.LONG	0	
						00000000	01C54	P.ANL: .BLKB	0	
						00000000	01C54	.LONG	0	
						02	01C58	P.ANM: .BLKB	0	
						01	01C59	P.ANN: .BYTE	2	
						0008	01C5A	.WORD	8	
						03	01C5C	.BYTE	3	
						03	01C5D	.BYTE	3	
						000A	01C5E	.WORD	10	
						09	01C60	.BYTE	9	
						0F	01C61	.BYTE	15	
						000A	01C62	.WORD	10	
						05	01C64	.BYTE	5	
						04	01C65	.BYTE	4	
						0011	01C66	.WORD	17	
						04	01C68	.BYTE	4	
						07	01C69	.BYTE	7	
						0016	01C6A	.WORD	22	
						08	01C6C	.BYTE	8	
						0A	01C6D	.BYTE	10	

	0011	01C6E	.WORD	17	
	01	01C70	.BYTE	1	
	0C	01C71	.BYTE	12	
	001B	01C72	.WORD	27	
	00000000	01C74	.LONG	0	
	01	01C78	.BYTE	1	
	02	01C79	.BYTE	2	
	001B	01C7A	.WORD	27	
	00000000	01C7C	.LONG	0	
	03	01C80	.BYTE	3	
	0E	01C81	.BYTE	14	
	000A	01C82	.WORD	10	
	09	01C84	.BYTE	9	
	0F	01C85	.BYTE	15	
	000A	01C86	.WORD	10	
	05	01C88	.BYTE	5	
	10	01C89	.BYTE	16	
	0011	01C8A	.WORD	17	
	04	01C8C	.BYTE	4	
	12	01C8D	.BYTE	18	
	0016	01C8E	.WORD	22	
	08	01C90	.BYTE	8	
	15	01C91	.BYTE	21	
	0011	01C92	.WORD	17	
	01	01C94	.BYTE	1	
	17	01C95	.BYTE	23	
	001B	01C96	.WORD	27	
	00000000	01C98	.LONG	0	
	05	01C9C	.BYTE	5	
	19	01C9D	.BYTE	25	
	0011	01C9E	.WORD	17	
	04	01CA0	.BYTE	4	
	1C	01CA1	.BYTE	28	
	0016	01CA2	.WORD	22	
	08	01CA4	.BYTE	8	
	1F	01CA5	.BYTE	31	
	0011	01CA6	.WORD	17	
	01	01CA8	.BYTE	1	
	21	01CA9	.BYTE	33	
	001B	01CAA	.WORD	27	
	00000000	01CAC	.LONG	0	
	05	01CB0	.BYTE	5	
	24	01CB1	.BYTE	36	
	0011	01CB2	.WORD	17	
	04	01CB4	.BYTE	4	
	26	01CB5	.BYTE	38	
	0016	01CB6	.WORD	22	
	08	01CB8	.BYTE	8	
	28	01CB9	.BYTE	40	
	0011	01CBA	.WORD	17	
	01	01CBC	.BYTE	1	
	2A	01CBD	.BYTE	42	
	001B	01CBE	.WORD	27	
	00000000	01CC0	.LONG	0	
	00000000	01CC4	.LONG	0	
00001BC1	00001BD5	00000C8D	00001AD1	0000197D	00001931
00000000	00000001	00000000	00000001	00001BD5	00001BD1
				01CE0	
			P.AND:	.LONG	6449, 6525, 6865, 3213, 7125, 7105, 7121, -
					7125, 1, 0, 1, 0, 0

3400021F	3300021F	3200021F	3100021F	3000021F	2D0008852	00000000	01CF8			
3C04A800	3900021F	3800021F	3700021F	3600021F	3500021F	00000010	01CFC			
		24000007	5F000007	3D042800	3E042800	01D00	01D18	P.ANP:	.LONG	16
									.LONG	755009618, 805306911, 822084127, -
										838861343, 855638559, 872415775, -
										889192991, 905970207, 922747423, -
										939524639, 956301855, 1006938112, -
										1040459776, 1023682560, 1593835527, -
										603979783
						01 03	01D40	P.ANR:	.BYTE	3, 1
						0005	01D42		.WORD	5
00	00	00	00	00	00	00 00	01D44		.BYTE	0, 0, 0, 0, 0, 0, 0, 0
					46	4F 02	01D4C		.ASCII	<2>\OF\
						01 03	01D4F	P.ANS:	.BYTE	3, 1
						0005	01D51		.WORD	5
00	00	00	00	00	00	00 00	01D53		.BYTE	0, 0, 0, 0, 0, 0, 0, 0
					4E	49 02	01D5B		.ASCII	<2>\IN\
						00 02	01D5E	P.ANT:	.BYTE	2, 0
						0017	01D60		.WORD	23
00	00	00	00	00	00	C8 19	01D62		.BYTE	25, -56, 0, 0, 0, 0, 0, 0
					54	4F 03	01D6A		.ASCII	<3>\NOT\
						00 03	01D6E	P.ANU:	.BYTE	3, 0
						0018	01D70		.WORD	24
00	00	00	00	00	00	14 14	01D72		.BYTE	20, 20, 0, 0, 0, 0, 0, 0
					44	4E 03	01D7A		.ASCII	<3>\AND\
						00 03	01D7E	P.ANV:	.BYTE	3, 0
						0019	01D80		.WORD	25
00	00	00	00	00	00	0A 0A	01D82		.BYTE	10, 10, 0, 0, 0, 0, 0, 0
					52	4F 02	01D8A		.ASCII	<2>\OR\
						02 03	01D8D	P.ANW:	.BYTE	3, 2
						002C	01D8F		.WORD	44
00	00	00	00	00	00	1E 1E	01D91		.BYTE	30, 30, 0, 0, 0, 0, 0, 0
					54	4F 03	01D99		.ASCII	<3>\NOT\
							01D9D		.BLKB	3
						00000006	01DA0		.LONG	6
00001D0A	00001CFB	00001CEB	00001CDB	00001CCC	00001CBD	00 03	01DA4	P.ANQ:	.LONG	7357, 7372, 7387, 7403, 7419, 7434
						000E	01DBC	P.ANX:	.BYTE	3, 0
						000E	01DBE		.WORD	14
00	00	00	00	00	00	1E 1E	01DC0		.BYTE	30, 30, 0, 0, 0, 0, 0, 0
					3D	20 54	01DC8		.ASCII	<5>\NOT =\
						4E 05	01DCE	P.ANY:	.BYTE	3, 0
						00 03	01DD0		.WORD	21
00	00	00	00	00	00	1E 1E	01DD2		.BYTE	30, 30, 0, 0, 0, 0, 0, 0
					3E	20 54	01DDA		.ASCII	<5>\NOT >\
						00 03	01DE0	P.ANZ:	.BYTE	3, 0
						0011	01DE2		.WORD	17
00	00	00	00	00	00	1E 1E	01DE4		.BYTE	30, 30, 0, 0, 0, 0, 0, 0
					3C	20 54	01DEC		.ASCII	<

00	00	00	00	00	00	00	00	01E12	.BYTE	0, 0, 0, 0, 0, 0, 0, 0
						28	01	01E1A	.ASCII	<1>\{\
						02	02	01E1C	P.AOE: .BYTE	2, 2
						003F	01E1E	.WORD	64	
00	00	00	00	00	00	C8	1E	01E20	.BYTE	30, -56, 0, 0, 0, 0, 0, 0
						3E	01	01E28	.ASCII	<1>\>\
						02	02	01E2A	P.AOF: .BYTE	2, 2
						0040	01E2C	.WORD	64	
00	00	00	00	00	00	C8	1E	01E2E	.BYTE	30, -56, 0, 0, 0, 0, 0, 0
						3C	01	01E36	.ASCII	<1>\<\
						02	02	01E38	P.AOG: .BYTE	2, 2
						0041	01E3A	.WORD	65	
00	00	00	00	00	00	C8	1E	01E3C	.BYTE	30, -56, 0, 0, 0, 0, 0, 0
						3D	01	01E44	.ASCII	<1>\=\
						02	02	01E46	P.AOH: .BYTE	2, 2
						000B	01E48	.WORD	11	
00	00	00	00	00	00	C8	05	01E4A	.BYTE	5, -56, 0, 0, 0, 0, 0, 0
						28	01	01E52	.ASCII	<1>\{\
						02	04	01E54	P.AOI: .BYTE	4, 2
						000C	01E56	.WORD	12	
00	00	00	00	00	00	06	C8	01E58	.BYTE	-56, 6, 0, 0, 0, 0, 0, 0
						29	01	01E60	.ASCII	<1>\)\
						00	03	01E62	P.AOJ: .BYTE	3, 0
						0008	01E64	.WORD	8	
00	00	00	00	00	00	3C	3C	01E66	.BYTE	60, 60, 0, 0, 0, 0, 0, 0
						2A	01	01E6E	.ASCII	<1>*\
						00	03	01E70	P.AOK: .BYTE	3, 0
						0009	01E72	.WORD	9	
00	00	00	00	00	00	3C	3C	01E74	.BYTE	60, 60, 0, 0, 0, 0, 0, 0
						2F	01	01E7C	.ASCII	<1>\/\
						00	02	01E7E	P.AOL: .BYTE	2, 0
						0004	01E80	.WORD	4	
00	00	00	00	00	00	C8	32	01E82	.BYTE	50, -56, 0, 0, 0, 0, 0, 0
						2B	01	01E8A	.ASCII	<1>\+\
						00	02	01E8C	P.AOM: .BYTE	2, 0
						0005	01E8E	.WORD	5	
00	00	00	00	00	00	C8	32	01E90	.BYTE	50, -56, 0, 0, 0, 0, 0, 0
						2D	01	01E98	.ASCII	<1>\-\
						00	03	01E9A	P.AON: .BYTE	3, 0
						0006	01E9C	.WORD	6	
00	00	00	00	00	00	28	28	01E9E	.BYTE	40, 40, 0, 0, 0, 0, 0, 0
						2B	01	01EA6	.ASCII	<1>\+\
						00	03	01EA8	P.AOO: .BYTE	3, 0
						0007	01EAA	.WORD	7	
00	00	00	00	00	00	28	28	01EAC	.BYTE	40, 40, 0, 0, 0, 0, 0, 0
						2D	01	01EB4	.ASCII	<1>\-\
						00	03	01EB6	P.AOP: .BYTE	3, 0
						000F	01EB8	.WORD	15	
00	00	00	00	00	00	1E	1E	01EBA	.BYTE	30, 30, 0, 0, 0, 0, 0, 0
						3E	01	01EC2	.ASCII	<1>\>\
						00	03	01EC4	P.AOQ: .BYTE	3, 0
						0013	01EC6	.WORD	19	
00	00	00	00	00	00	1E	1E	01EC8	.BYTE	30, 30, 0, 0, 0, 0, 0, 0
						3C	01	01ED0	.ASCII	<1>\<\
						00	03	01ED2	P.AOR: .BYTE	3, 0
						000D	01ED4	.WORD	13	
00	00	00	00	00	00	1E	1E	01ED6	.BYTE	30, 30, 0, 0, 0, 0, 0, 0

00001DB5	00001DA7	00001D99	00001D8B	00001D7D	00000011	01EDE	.ASCII	<1>\=\	
00001E09	00001DFB	00001DED	00001DDF	00001DD1	00001D6F	01EE0	.LONG	17	
	00001E4F	00001E41	00001E33	00001E25	00001DC3	01EE4	.LONG	7535, 7549, 7563, 7577, 7591, 7605, 7619, -	
					00001E17	01EFC		7633, 7647, 7661, 7675, 7689, 7703, 7717, -	
					00 00	01F14	P.AOT:	.BYTE	0, 0
					0002	01F28		.WORD	2
				00 00	00 00	01F2C		.BYTE	0, 0, 0, 0
				29 01	01F30	01F30	P.AOU:	.ASCII	<f>\;\
				02 00	01F32	01F32		.BYTE	0, 2
				0003	01F34	01F34		.WORD	3
				00 00	00 00	01F36		.BYTE	0, 0, 0, 0
				3A 01	01F3A	01F3A	P.AOV:	.ASCII	<f>\:\
				00 00	01F3C	01F3C		.BYTE	0, 0
				0001	01F3E	01F3E		.WORD	1
				00 00	00 00	01F40		.BYTE	0, 0, 0, 0
				2C 01	01F44	01F44		.ASCII	<f>\,\
					01F46	01F46		.BLKB	2
					00000003	01F48		.LONG	3
				00001EB9	00001EAF	01F4C	P.AOS:	.LONG	7845, 7855, 7865
					00000000	01F58		.LONG	0
					00000000	01F5C	P.AOW:	.BLKB	0
					00000000	01F5C		.LONG	0
					01	01F60	P.AOX:	.BLKB	0
					0E	01F60	P.AOY:	.BYTE	1
					0005	01F61		.BYTE	14
					03	01F62		.WORD	5
					01	01F64		.BYTE	3
					0003	01F65		.BYTE	1
					00	01F66		.WORD	3
					0A	01F68		.BYTE	0
					0021	01F69		.BYTE	10
					01	01F6A		.WORD	33
					0F	01F6C		.BYTE	1
					0012	01F6D		.BYTE	15
					00	01F6E		.WORD	18
					0A	01F70		.BYTE	0
					0021	01F71		.BYTE	10
					01	01F72		.WORD	33
					0E	01F74		.BYTE	1
					0005	01F75		.BYTE	14
					03	01F76		.WORD	5
					01	01F78		.BYTE	3
					0012	01F79		.BYTE	1
					02	01F7A		.WORD	18
					0D	01F7C		.BYTE	2
					000C	01F7D		.BYTE	13
					06	01F7E		.WORD	12
					0D	01F80		.BYTE	6
					000C	01F81		.BYTE	13
					07	01F82		.WORD	12
					0D	01F84		.BYTE	7
					000C	01F85		.BYTE	13
					08	01F86		.WORD	12
					0D	01F88		.BYTE	8
					000C	01F89		.BYTE	13
						01F8A		.WORD	12

00	01F8C	.BYTE	0
0C	01F8D	.BYTE	12
0021	01F8E	.WORD	33
01	01F90	.BYTE	1
01	01F91	.BYTE	1
000C	01F92	.WORD	12
02	01F94	.BYTE	2
01	01F95	.BYTE	1
000C	01F96	.WORD	12
06	01F98	.BYTE	6
01	01F99	.BYTE	1
000C	01F9A	.WORD	12
07	01F9C	.BYTE	7
01	01F9D	.BYTE	1
000C	01F9E	.WORD	12
08	01FA0	.BYTE	8
01	01FA1	.BYTE	1
000C	01FA2	.WORD	12
00	01FA4	.BYTE	0
0C	01FA5	.BYTE	12
0021	01FA6	.WORD	33
01	01FA8	.BYTE	1
0F	01FA9	.BYTE	15
0012	01FAA	.WORD	18
03	01FAC	.BYTE	3
08	01FAD	.BYTE	8
0021	01FAE	.WORD	33
08	01FB0	.BYTE	8
05	01FB1	.BYTE	5
0019	01FB2	.WORD	25
07	01FB4	.BYTE	7
06	01FB5	.BYTE	6
0019	01FB6	.WORD	25
0D	01FB8	.BYTE	13
12	01FB9	.BYTE	18
0019	01FBA	.WORD	25
09	01FBC	.BYTE	9
07	01FBD	.BYTE	7
0019	01FBE	.WORD	25
00	01FC0	.BYTE	0
10	01FC1	.BYTE	16
0021	01FC2	.WORD	33
01	01FC4	.BYTE	1
01	01FC5	.BYTE	1
001F	01FC6	.WORD	31
04	01FC8	.BYTE	4
01	01FC9	.BYTE	1
001D	01FCA	.WORD	29
05	01FCC	.BYTE	5
01	01FCD	.BYTE	1
001D	01FCE	.WORD	29
00	01FD0	.BYTE	0
08	01FD1	.BYTE	8
0021	01FD2	.WORD	33
01	01FD4	.BYTE	1
01	01FD5	.BYTE	1
001F	01FD6	.WORD	31

.....

1 5
16-Sep-1984 02:10:13
14-Sep-1984 12:17:30VAX-11 Bliss-32 V4.0-742
[DEBUG.SRC]DBGPARSER.B32;1Page 101
(17)

00	01FD8	.BYTE	0
09	01FD9	.BYTE	9
0021	01FDA	.WORD	33
01	01FDC	.BYTE	1
01	01FDD	.BYTE	1
001F	01FDE	.WORD	31
00	01FE0	.BYTE	0
09	01FE1	.BYTE	9
0021	01FE2	.WORD	33
00	01FE4	.BYTE	0
08	01FE5	.BYTE	11
0021	01FE6	.WORD	33
02	01FE8	.BYTE	2
01	01FE9	.BYTE	1
0007	01FEA	.WORD	7
03	01FEC	.BYTE	3
03	01FED	.BYTE	3
0009	01FEE	.WORD	9
09	01FF0	.BYTE	9
0F	01FF1	.BYTE	15
0009	01FF2	.WORD	9
05	01FF4	.BYTE	5
06	01FF5	.BYTE	6
000E	01FF6	.WORD	14
04	01FF8	.BYTE	4
09	01FF9	.BYTE	9
0013	01FFA	.WORD	19
01	01FFC	.BYTE	1
0C	01FFD	.BYTE	12
0018	01FFE	.WORD	24
00000000	02000	.LONG	0
01	02004	.BYTE	1
02	02005	.BYTE	2
0018	02006	.WORD	24
00000000	02008	.LONG	0
03	0200C	.BYTE	3
0E	0200D	.BYTE	14
0009	0200E	.WORD	9
09	02010	.BYTE	9
0F	02011	.BYTE	15
0009	02012	.WORD	9
04	02014	.BYTE	4
13	02015	.BYTE	19
0013	02016	.WORD	19
01	02018	.BYTE	1
17	02019	.BYTE	23
0018	0201A	.WORD	24
00000000	0201C	.LONG	0
03	02020	.BYTE	3
18	02021	.BYTE	24
0009	02022	.WORD	9
05	02024	.BYTE	5
1B	02025	.BYTE	27
000E	02026	.WORD	14
04	02028	.BYTE	4
1E	02029	.BYTE	30
0013	0202A	.WORD	19

P.A02:

						01	0202C		.BYTE	1	
						23	0202D		.BYTE	35	
					0018	0202E		.WORD	24		
					00000000	02030		.LONG	0		
					04	02034		.BYTE	4		
					27	02035		.BYTE	39		
					0016	02036		.WORD	22		
					01	02038		.BYTE	1		
					28	02039		.BYTE	43		
					0018	0203A		.WORD	24		
					00000000	0203C		.LONG	0		
					01	02040		.BYTE	1		
					28	02041		.BYTE	43		
					0018	02042		.WORD	24		
					00000000	02044		.LONG	0		
					00000000	02048		.LONG	0		
00001EC9	00001F65	00001EDD	00001E61	00001D21	00001C7D	0204C	P.APA:	.LONG	7293, 7457, 7777, 7901, 8037, 7881, 7897, -		
00000001	00000000	00000001	00000000	00001EDD	00001ED9	02064			7901, 0, 1, 0, 1, 0		
					00000000	0207C					
					00000002	02080		.LONG	2		
				2F002800	2E010038	02084	P.APB:	.LONG	771817528, 788539392		
					00000000	0208C		.LONG	0		
						02090	P.APC:	.BLKB	0		
					01 02	02090	P.APE:	.BYTE	2, 1		
					0002	02092		.WORD	2		
00	00	00	00	00	00	00 00		.BYTE	0, 0, 0, 0, 0, 0, 0, 0		
					5C 01	02094		.ASCII	<1><92>		
					01 03	0209E	P.APF:	.BYTE	3, 1		
					0003	020A0		.WORD	3		
00	00	00	00	00	00	00 00		.BYTE	0, 0, 0, 0, 0, 0, 0, 0		
					5C 01	020A2		.ASCII	<1><92>		
					01 04	020AA	P.APG:	.BYTE	4, 1		
					0004	020AC		.WORD	4		
00	00	00	00	00	00	00 00		.BYTE	0, 0, 0, 0, 0, 0, 0, 0		
					28 01	020B0		.ASCII	<1>\(\		
					01 03	020B8	P.APH:	.BYTE	3, 1		
					0005	020BA		.WORD	5		
00	00	00	00	00	00	00 00		.BYTE	0, 0, 0, 0, 0, 0, 0, 0		
					2E 01	020BE		.ASCII	<1>\.\		
					00 03	020C6	P.API:	.BYTE	3, 0		
					0006	020C8		.WORD	6		
00	00	00	00	00	00	3C 3C		.BYTE	60, 60, 0, 0, 0, 0, 0, 0		
					2B 01	020CA		.ASCII	<1>\+\		
					00 03	020CC	P.APJ:	.BYTE	3, 0		
					0007	020D4		.WORD	7		
00	00	00	00	00	00	3C 3C		.BYTE	60, 60, 0, 0, 0, 0, 0, 0		
					2D 01	020D6		.ASCII	<1>\-\		
					00 02	020D8	P.APK:	.BYTE	2, 0		
					0004	020DA		.WORD	4		
00	00	00	00	00	00	C8 46		.BYTE	70, -56, 0, 0, 0, 0, 0, 0		
					2B 01	020E2		.ASCII	<1>\+\		
					00 02	020E4	P.APL:	.BYTE	2, 0		
					0005	020E6		.WORD	5		
00	00	00	00	00	00	C8 46		.BYTE	70, -56, 0, 0, 0, 0, 0, 0		
					2D 01	020E8		.ASCII	<1>\-\		
					00 03	020F0	P.APM:	.BYTE	3, 0		
					0008	020F2		.WORD	8		
						020F4		.BYTE			
						020F6		.ASCII			
						020FE		.WORD			
						02100		.BYTE			
						02102		.WORD			

00	00	00	00	00	00	50	50	02104	.BYTE	80, 80, 0, 0, 0, 0, 0, 0	
						2A	01	0210C	.ASCII	<1>*\	
						00	03	0210E	P.APN: .BYTE	3, 0	
						0009		02110	.WORD	9	
00	00	00	00	00	00	50	50	02112	.BYTE	80, 80, 0, 0, 0, 0, 0, 0	
						2F	01	0211A	.ASCII	<1>\/\	
						00	03	0211C	P.APD: .BYTE	3, 0	
						000A		0211E	.WORD	10	
00	00	00	00	00	00	5C	5A	02120	.BYTE	90, 92, 0, 0, 0, 0, 0, 0	
						2A	02	02128	.ASCII	<2>**\	
						00	03	0212B	P.APP: .BYTE	3, 0	
						0023		0212D	.WORD	35	
00	00	00	00	00	00	3C	3C	0212F	.BYTE	60, 60, 0, 0, 0, 0, 0, 0	
						2F	02	02137	.ASCII	<2>\//\	
						02	02	0213A	P.APD: .BYTE	2, 2	
						000B		0213C	.WORD	11	
00	00	00	00	00	00	C8	05	0213E	.BYTE	5, -56, 0, 0, 0, 0, 0, 0	
						28	01	02146	.ASCII	<1>\(\	
						02	04	02148	P.APR: .BYTE	4, 2	
						000C		0214A	.WORD	12	
00	00	00	00	00	00	06	C8	0214C	.BYTE	-56, 6, 0, 0, 0, 0, 0, 0	
						29	01	02154	.ASCII	<1>\)\	
								02156	.BLKB	2	
						0000000E		02158	.LONG	14	
00002053	00002045	00002037	00002029	0000201B	0000200D	00002000		0215C	P.APD: .LONG	8205, 8219, 8233, 8247, 8261, 8275, 8289, -	
000020A8	00002099	0000208B	0000207D	0000206F	00002061	00002055		02174		8303, 8317, 8331, 8345, 8360, 8375, 8389	
						000020B7		0218C			
						00	02	02194	P.APS: .BYTE	2, 0	
						0003		02196	.WORD	3	
00	00	00	00	00	00	C8	28	02198	.BYTE	40, -56, 0, 0, 0, 0, 0, 0	
						2E	01	021A0	.ASCII	<1>\.\	
						01	03	021A2	P.APT: .BYTE	3, 1	
						0005		021A4	.WORD	5	
00	00	00	00	00	00	00	00	021A6	.BYTE	0, 0, 0, 0, 0, 0, 0, 0	
						2E	01	021AE	.ASCII	<1>\.\	
						00	03	021B0	P.APV: .BYTE	3, 0	
						000D		021B2	.WORD	13	
00	00	00	00	00	00	32	32	021B4	.BYTE	50, 50, 0, 0, 0, 0, 0, 0	
						2E	04	021BC	.ASCII	<4>\.EQ.\	
						00	03	021C1	P.APW: .BYTE	3, 0	
						000E		021C3	.WORD	14	
00	00	00	00	00	00	32	32	021C5	.BYTE	50, 50, 0, 0, 0, 0, 0, 0	
						2E	04	021CD	.ASCII	<4>\.NE.\	
						00	03	021D2	P.APX: .BYTE	3, 0	
						000F		021D4	.WORD	15	
00	00	00	00	00	00	32	32	021D6	.BYTE	50, 50, 0, 0, 0, 0, 0, 0	
						2E	04	021DE	.ASCII	<4>\.Gf.\	
						00	03	021E3	P.APY: .BYTE	3, 0	
						0011		021E5	.WORD	17	
00	00	00	00	00	00	32	32	021E7	.BYTE	50, 50, 0, 0, 0, 0, 0, 0	
						2E	04	021EF	.ASCII	<4>\.GE.\	
						00	03	021F4	P.APZ: .BYTE	3, 0	
						0013		021F6	.WORD	19	
00	00	00	00	00	00	32	32	021F8	.BYTE	50, 50, 0, 0, 0, 0, 0, 0	
						2E	04	02200	.ASCII	<4>\.Lf.\	
						00	03	02205	P.AQA: .BYTE	3, 0	
						0015		02207	.WORD	21	

00	00	00	00	00	00	32	32	02209	.BYTE	50	50	0	0	0	0	0	0	
			2E	45	4C	2E	04	02211	.ASCII	<4>	\.LE.							
						00	02	02216	P.AQB:	.BYTE	2	0						
						0017		02218	.WORD	2								
00	00	00	00	00	00	C8	28	0221A	.BYTE	40	-56	0	0	0	0	0	0	
			2E	54	4F	4E	2E	05	02222	.ASCII	<5>	\.NOT.						
						00	03	02228	P.AQC:	.BYTE	3	0						
						0018		0222A	.WORD	3								
00	00	00	00	00	00	1E	1E	0222C	.BYTE	30	30	0	0	0	0	0	0	
			2E	44	4E	41	2E	05	02234	.ASCII	<5>	\.AND.						
						00	03	0223A	P.AQD:	.BYTE	3	0						
						0019		0223C	.WORD	2								
00	00	00	00	00	00	14	14	0223E	.BYTE	20	20	0	0	0	0	0	0	
			2E	52	4F	2E	04	02246	.ASCII	<4>	\.OR.							
						00	03	0224B	P.AQE:	.BYTE	3	0						
						001A		0224D	.WORD	2								
00	00	00	00	00	00	0A	0A	0224F	.BYTE	10	10	0	0	0	0	0	0	
			2E	52	4F	58	2E	05	02257	.ASCII	<5>	\.XOR.						
						00	03	0225D	P.AQF:	.BYTE	3	0						
						001B		0225F	.WORD	2								
00	00	00	00	00	00	0A	0A	02261	.BYTE	10	10	0	0	0	0	0	0	
			2E	56	51	45	2E	05	02269	.ASCII	<5>	\.EQV.						
						00	03	0226F	P.AQG:	.BYTE	3	0						
						001A		02271	.WORD	2								
00	00	00	00	00	00	0A	0A	02273	.BYTE	10	10	0	0	0	0	0	0	
			2E	56	51	45	2E	06	0227B	.ASCII	<6>	\.NEQV.						
								02282	.BLKB	2								
						0000000C		02284	.LONG	12								
00002182	00002171	00002160	0000214F	0000213E	0000212D	00002120		02288	P.APU:	.LONG	8493	8510	8527	8544	8561	8578	8595	-
000021EC	000021DA	000021C8	000021B7	000021A5	00002193	00002180		022A0			8613	8631	8648	8666	8684			
						01	00	022B8	P.AQI:	.BYTE	0	1						
						0002		022BA	.WORD	2								
				00	00	00	00	022BC	.BYTE	0	0	0	0					
						29	01	022C0	.ASCII	<1>	\.;							
						00	00	022C2	P.AQJ:	.BYTE	0	0						
						0001		022C4	.WORD	1								
				00	00	00	00	022C6	.BYTE	0	0	0	0					
						2C	01	022CA	.ASCII	<1>	\.;							
						00	00	022CC	P.AQK:	.BYTE	0	0						
						0003		022CE	.WORD	3								
				00	00	00	00	022D0	.BYTE	0	0	0	0					
						3A	01	022D4	.ASCII	<1>	\.;							
								022D6	.BLKB	2								
						00000003		022D8	.LONG	3								
00002249	0000223F	00002235				00002235		022DC	P.AQH:	.LONG	8757	8767	8777					
				00	02	08	01	022E8	P.AQM:	.BYTE	1	8	2	0				
						00000001		022EC	.LONG	1								
	2E	45	55	52	54	2E	06	022F0	.ASCII	<6>	\.TRUE.							
				00	02	08	01	022F7	P.AQN:	.BYTE	1	8	2	0				
						00000000		022FB	.LONG	0								
2E	45	53	4C	41	46	2E	07	022FF	.ASCII	<7>	\.FALSE.							
								02307	.BLKB	1								
						00000002		02308	.LONG	2								
						00002274		0230C	P.AQL:	.LONG	8805	8820						
						00000000		02314	.LONG	0								
								02318	P.AQO:	.BLKB	0							
						02		02318	P.AQP:	.BYTE	2							

01	02319	.BYTE	1
0007	0231A	.WORD	7
03	0231C	.BYTE	3
03	0231D	.BYTE	3
0009	0231E	.WORD	9
09	02320	.BYTE	9
0F	02321	.BYTE	15
0009	02322	.WORD	9
05	02324	.BYTE	5
04	02325	.BYTE	4
000E	02326	.WORD	14
04	02328	.BYTE	4
07	02329	.BYTE	7
0012	0232A	.WORD	18
01	0232C	.BYTE	1
0C	0232D	.BYTE	12
0016	0232E	.WORD	22
00000000	02330	.LONG	0
01	02334	.BYTE	1
02	02335	.BYTE	2
0016	02336	.WORD	22
00000000	02338	.LONG	0
03	0233C	.BYTE	3
0E	0233D	.BYTE	14
0009	0233E	.WORD	9
09	02340	.BYTE	9
0F	02341	.BYTE	15
0009	02342	.WORD	9
04	02344	.BYTE	4
12	02345	.BYTE	18
0012	02346	.WORD	18
01	02348	.BYTE	1
17	02349	.BYTE	23
0016	0234A	.WORD	22
00000000	0234C	.LONG	0
05	02350	.BYTE	5
19	02351	.BYTE	25
000E	02352	.WORD	14
04	02354	.BYTE	4
1C	02355	.BYTE	28
0012	02356	.WORD	18
01	02358	.BYTE	1
21	02359	.BYTE	33
0016	0235A	.WORD	22
00000000	0235C	.LONG	0
05	02360	.BYTE	5
24	02361	.BYTE	36
000E	02362	.WORD	14
04	02364	.BYTE	4
26	02365	.BYTE	38
0012	02366	.WORD	18
01	02368	.BYTE	1
2A	02369	.BYTE	42
0016	0236A	.WORD	22
00000000	0236C	.LONG	0
00000000	02370	.LONG	0
00002001	02374	.LONG	8193, 8205, 8409, 3213, 8853, 8793, 8841, -

00002259 00002295 00000C8D 000020D9 0000200D 00002001 P.AQQ:

						00 03	02496	P.ARG:	.BYTE	3 0	
						001F	02498		.WORD	31	
00	00	00	00	00	00	1E 1E	0249A		.BYTE	30, 30, 0, 0, 0, 0, 0, 0	
				44	4E	41 03	024A2		.ASCII	<3>\AND\	
						00 03	024A6	P.ARH:	.BYTE	3 0	
						0020	024A8		.WORD	32	
00	00	00	00	00	00	14 14	024AA		.BYTE	20, 20, 0, 0, 0, 0, 0, 0	
					52	4F 02	024B2		.ASCII	<2>\OR\	
						00 03	024B5	P.ARI:	.BYTE	3 0	
						0022	024B7		.WORD	34	
00	00	00	00	00	00	0A 0A	024B9		.BYTE	10, 10, 0, 0, 0, 0, 0, 0	
				56	51	45 03	024C1		.ASCII	<3>\EQV\	
						00 03	024C5	P.ARJ:	.BYTE	3 0	
						0021	024C7		.WORD	33	
00	00	00	00	00	00	0A 0A	024C9		.BYTE	10, 10, 0, 0, 0, 0, 0, 0	
					52	4F 03	024D1		.ASCII	<3>\XOR\	
						00 03	024D5	P.ARK:	.BYTE	3 0	
						0025	024D7		.WORD	37	
00	00	00	00	00	00	46 46	024D9		.BYTE	70, 70, 0, 0, 0, 0, 0, 0	
					44	4F 03	024E1		.ASCII	<3>\MOD\	
							024E5		.BLKB	3	
						00000012	024E8		.LONG	18	
0000238F	0000237F	0000236E	0000235E	0000234D	0000233D	00002330	024EC	P.AQS:	.LONG	9021, 9037, 9054, 9070, 9087, 9103, 9120, -	
000023F2	000023E2	000023D1	000023C1	000023B0	000023A0	000023A0	02504			9136, 9153, 9169, 9186, 9202, 9219, 9235, -	
00002452	00002442	00002432	00002423	00002413	00002403	00002403	0251C			9251, 9266, 9282, 9298	
						01 02	02534	P.ARM:	.BYTE	2, 1	
						0002	02536		.WORD	2	
00	00	00	00	00	00	00 00	02538		.BYTE	0 0 0, 0, 0, 0, 0, 0	
						5C 01	02540		.ASCII	<1><92>	
						01 03	02542	P.ARN:	.BYTE	3, 1	
						0003	02544		.WORD	3	
00	00	00	00	00	00	00 00	02546		.BYTE	0 0 0, 0, 0, 0, 0, 0	
						5C 01	0254E		.ASCII	<1><92>	
						02 02	02550	P.ARO:	.BYTE	2 2	
						0008	02552		.WORD	11	
00	00	00	00	00	00	C8 05	02554		.BYTE	5 -56, 0, 0, 0, 0, 0, 0	
						28 01	0255C		.ASCII	<1>\(\	
						02 04	0255E	P.ARP:	.BYTE	4, 2	
						000C	02560		.WORD	12	
00	00	00	00	00	00	06 C8	02562		.BYTE	-56, 6, 0, 0, 0, 0, 0, 0	
						29 01	0256A		.ASCII	<1>\)\	
						02 04	0256C	P.ARQ:	.BYTE	4, 2	
						0031	0256E		.WORD	49	
00	00	00	00	00	00	6E C8	02570		.BYTE	-56, 110, 0, 0, 0, 0, 0, 0	
						3C 01	02578		.ASCII	<1>\(<\	
						00 02	0257A	P.ARR:	.BYTE	2, 0	
						0003	0257C		.WORD	3	
00	00	00	00	00	00	C8 64	0257E		.BYTE	100, -56, 0, 0, 0, 0, 0, 0	
						2E 01	02586		.ASCII	<1>\.\	
						00 02	02588	P.ARS:	.BYTE	2, 0	
						0003	0258A		.WORD	3	
00	00	00	00	00	00	C8 64	0258C		.BYTE	100, -56, 0, 0, 0, 0, 0, 0	
						40 01	02594		.ASCII	<1>\a\	
						00 02	02596	P.ART:	.BYTE	2, 0	
						0004	02598		.WORD	4	
00	00	00	00	00	00	C8 5A	0259A		.BYTE	90, -56, 0, 0, 0, 0, 0, 0	
						2B 01	025A2		.ASCII	<1>\+\	

					00 02	025A4	P.ARU:	.BYTE	2, 0	
					0005	025A6		.WORD	5	
00	00	00	00	00	00	C8 5A	025A8	.BYTE	90, -56, 0, 0, 0, 0, 0, 0	
					2D 01	025B0		.ASCII	<1>\-\\	
					00 03	025B2	P.ARV:	.BYTE	3, 0	
					0026	025B4		.WORD	38	
00	00	00	00	00	00	50 50	025B6	.BYTE	80, 80, 0, 0, 0, 0, 0, 0	
					40 01	025B8		.ASCII	<1>\@\\	
					00 03	025C0	P.ARW:	.BYTE	3, 0	
					0008	025C2		.WORD	8	
00	00	00	00	00	00	46 46	025C4	.BYTE	70, 70, 0, 0, 0, 0, 0, 0	
					2A 01	025C6		.ASCII	<1>*\\	
					00 03	025C8	P.ARX:	.BYTE	3, 0	
					0009	025D0		.WORD	9	
00	00	00	00	00	00	46 46	025D2	.BYTE	70, 70, 0, 0, 0, 0, 0, 0	
					2F 01	025D4		.ASCII	<1>\//\\	
					00 03	025D6	P.ARY:	.BYTE	3, 0	
					0006	025D8		.WORD	6	
00	00	00	00	00	00	3C 3C	025E0	.BYTE	60, 60, 0, 0, 0, 0, 0, 0	
					2B 01	025E2		.ASCII	<1>\+\\	
					00 03	025E4	P.ARZ:	.BYTE	3, 0	
					0007	025E6		.WORD	7	
00	00	00	00	00	00	3C 3C	025E8	.BYTE	60, 60, 0, 0, 0, 0, 0, 0	
					2D 01	025F0		.ASCII	<1>\-\\	
					0000000E	025F2		.LONG	14	
000024F7	000024E9	000024DB	000024CD	000024BF	000024B1	025FC	P.ARL:	.LONG	9393, 9407, 9421, 9435, 9449, 9463, 9477, -	
0000254B	0000253D	0000252F	00002521	00002513	00002505	02614			9491, 9505, 9519, 9533, 9547, 9561, 9575	
				00002567	00002559	0262C				
					00000000	02634		.LONG	0	
						02638	P.ASA:	.BLKB	0	
					00000000	02638		.LONG	0	
						0263C	P.ASB:	.BLKB	0	
					00000000	0263C		.LONG	0	
						02640	P.ASC:	.BLKB	0	
					02	02640	P.ASD:	.BYTE	2	
					01	02641		.BYTE	1	
					0005	02642		.WORD	3	
					03	02644		.BYTE	3	
					03	02645		.BYTE	3	
					0007	02646		.WORD	7	
					09	02648		.BYTE	9	
					0F	02649		.BYTE	15	
					0007	0264A		.WORD	7	
					01	0264C		.BYTE	1	
					0C	0264D		.BYTE	12	
					000B	0264E		.WORD	11	
					00000000	02650		.LONG	0	
					01	02654		.BYTE	1	
					02	02655		.BYTE	2	
					000B	02656		.WORD	11	
					00000000	02658		.LONG	0	
					03	0265C		.BYTE	3	
					0E	0265D		.BYTE	14	
					0007	0265E		.WORD	7	
					09	02660		.BYTE	9	
					0F	02661		.BYTE	15	
					0007	02662		.WORD	7	

					01	02664	.BYTE	1	
					17	02665	.BYTE	23	
					000B	02666	.WORD	11	
					00000000	02668	.LONG	0	
					00000000	0266C	.LONG	0	
000025B5	000025BD	00000C8D	00002579	00002469	00002329	02670	P.ASE: .LONG	9001, 9321, 9593, 3213, 9661, 9653, 9657, -	
00000000	00000000	00000001	00000000	000025BD	000025B9	02688		9661, 0, 1, 0, 0, 0	
					00000000	026A0			
					00000007	026A4	.LONG	7	
5B000800	2E058838	5E010800	3E042800	3D042800	3C00A800	026A8	P.ASF: .LONG	1006675968, 1023682560, 1040459776, -	
					5D040000	026C0		1577125888, 772114488, 1526728704, -	
								1560543232	
					00 03	026C4	P.ASH: .BYTE	3 0	
					0030	026C6	.WORD	46	
					00 00 00 00 00 00 46 46	026C8	.BYTE	70, 70, 0, 0, 0, 0, 0, 0	
					56 49 44 03	026D0	.ASCII	<3>\DIV\	
					00 03	026D4	P.ASI: .BYTE	3, 0	
					0024	026D6	.WORD	36	
					00 00 00 00 00 00 46 46	026D8	.BYTE	70, 70, 0, 0, 0, 0, 0, 0	
					44 4F 4D 03	026E0	.ASCII	<3>\MOD\	
					00 03	026E4	P.ASJ: .BYTE	3, 0	
					0025	026E6	.WORD	37	
					00 00 00 00 00 00 46 46	026E8	.BYTE	70, 70, 0, 0, 0, 0, 0, 0	
					4D 45 52 03	026F0	.ASCII	<3>\REM\	
					00 03	026F4	P.ASK: .BYTE	3, 0	
					0018	026F6	.WORD	24	
					00 00 00 00 00 00 46 46	026F8	.BYTE	70, 70, 0, 0, 0, 0, 0, 0	
					44 4E 41 03	02700	.ASCII	<3>\AND\	
					00 03	02704	P.ASL: .BYTE	3, 0	
					0019	02706	.WORD	25	
					00 00 00 00 00 00 3C 3C	02708	.BYTE	60, 60, 0, 0, 0, 0, 0, 0	
					52 4F 02	02710	.ASCII	<2>\OR\	
					00 02	02713	P.ASM: .BYTE	2, 0	
					0017	02715	.WORD	23	
					00 00 00 00 00 00 C8 5A	02717	.BYTE	90, -56, 0, 0, 0, 0, 0, 0	
					54 4F 4E 03	0271F	.ASCII	<3>\NOT\	
					00 03	02723	P.ASN: .BYTE	3, 0	
					002A	02725	.WORD	42	
					00 00 00 00 00 00 32 32	02727	.BYTE	50, 50, 0, 0, 0, 0, 0, 0	
					4E 49 02	0272F	.ASCII	<2>\IN\	
						02732	.BLKB	2	
					00000007	02734	.LONG	7	
00002690	00002681	00002671	00002661	00002651	00002641	02738	P.ASG: .LONG	9793, 9809, 9825, 9841, 9857, 9872, 9888	
					000026A0	02750			
					01 02	02754	P.ASP: .BYTE	2, 1	
					0002	02756	.WORD	2	
					00 00 00 00 00 00 00 00	02758	.BYTE	0, 0, 0, 0, 0, 0, 0, 0	
					5C 01	02760	.ASCII	<1><92>	
					01 03	02762	P.ASQ: .BYTE	3, 1	
					0003	02764	.WORD	3	
					00 00 00 00 00 00 00 00	02766	.BYTE	0, 0, 0, 0, 0, 0, 0, 0	
					5C 01	0276E	.ASCII	<1><92>	
					01 04	02770	P.ASR: .BYTE	4, 1	
					0004	02772	.WORD	4	
					00 00 00 00 00 00 00 00	02774	.BYTE	0, 0, 0, 0, 0, 0, 0, 0	
					5B 01	0277C	.ASCII	<1>\t\	
					01 03	0277E	P.ASS: .BYTE	3, 1	

						0005	02780	.WORD	5
00	00	00	00	00	00	00 00	02782	.BYTE	0 0 0 0 0 0 0 0
						2E 01	0278A	.ASCII	<f>\.
						01 04	0278C P.AST:	.BYTE	4 1
						0007	0278E	.WORD	7
00	00	00	00	00	00	00 00	02790	.BYTE	0 0 0 0 0 0 0 0
						5E 01	02798	.ASCII	<f>\.
						01 04	0279A P.ASU:	.BYTE	4 1
						000A	0279C	.WORD	10
00	00	00	00	00	00	00 00	0279E	.BYTE	0 0 0 0 0 0 0 0
						28 01	027A6	.ASCII	<f>\(\
						02 02	027A8 P.ASV:	.BYTE	2 2
						000B	027AA	.WORD	11
00	00	00	00	00	00	C8 05	027AC	.BYTE	5 -56 0 0 0 0 0 0
						2B 01	027B4	.ASCII	<f>\(\
						02 04	027B6 P.ASW:	.BYTE	4 2
						000C	027B8	.WORD	12
00	00	00	00	00	00	06 C8	027BA	.BYTE	-56 6 0 0 0 0 0 0
						29 01	027C2	.ASCII	<f>\)\
						02 02	027C4 P.ASX:	.BYTE	2 2
						0039	027C6	.WORD	57
00	00	00	00	00	00	C8 05	027C8	.BYTE	5 -56 0 0 0 0 0 0
						5B 01	027D0	.ASCII	<f>\[\
						00 02	027D2 P.ASY:	.BYTE	2 0
						0004	027D4	.WORD	4
00	00	00	00	00	00	C8 3C	027D6	.BYTE	60 -56 0 0 0 0 0 0
						2B 01	027DE	.ASCII	<f>\+\
						00 02	027E0 P.ASZ:	.BYTE	2 0
						0005	027E2	.WORD	5
00	00	00	00	00	00	C8 3C	027E4	.BYTE	60 -56 0 0 0 0 0 0
						2D 01	027EC	.ASCII	<f>\-\
						00 03	027EE P.ATA:	.BYTE	3 0
						000A	027F0	.WORD	10
00	00	00	00	00	00	5D 50	027F2	.BYTE	80 80 0 0 0 0 0 0
						2A 02	027FA	.ASCII	<2>**\
						00 03	027FD P.ATB:	.BYTE	3 0
						000B	027FF	.WORD	8
00	00	00	00	00	00	46 46	02801	.BYTE	70 70 0 0 0 0 0 0
						2A 01	02809	.ASCII	<f>*\
						00 03	0280B P.ATC:	.BYTE	3 0
						0009	0280D	.WORD	9
00	00	00	00	00	00	46 46	0280F	.BYTE	70 70 0 0 0 0 0 0
						2F 01	02817	.ASCII	<f>\/\
						00 03	02819 P.ATD:	.BYTE	3 0
						0006	0281B	.WORD	6
00	00	00	00	00	00	3C 3C	0281D	.BYTE	60 60 0 0 0 0 0 0
						2B 01	02825	.ASCII	<f>\+\
						00 03	02827 P.ATE:	.BYTE	3 0
						0007	02829	.WORD	7
00	00	00	00	00	00	3C 3C	0282B	.BYTE	60 60 0 0 0 0 0 0
						2D 01	02833	.ASCII	<f>\-\
						00 03	02835 P.ATF:	.BYTE	3 0
						0013	02837	.WORD	19
00	00	00	00	00	00	32 32	02839	.BYTE	50 50 0 0 0 0 0 0
						3C 01	02841	.ASCII	<f>\<
						00 03	02843 P.ATG:	.BYTE	3 0
						0015	02845	.WORD	21

00	00	00	00	00	00	32	32	02847	.BYTE	50, 50, 0, 0, 0, 0, 0, 0	
					3D	3C	02	0284F	.ASCII	<2>\<=\	
					00	00	03	02852	P.ATH: .BYTE	3, 0	
						000F	02854	.WORD	15		
00	00	00	00	00	00	32	32	02856	.BYTE	50, 50, 0, 0, 0, 0, 0, 0	
						3E	01	0285E	.ASCII	<1>\>\	
					00	00	03	02860	P.ATI: .BYTE	3, 0	
						0011	02862	.WORD	17		
00	00	00	00	00	00	32	32	02864	.BYTE	50, 50, 0, 0, 0, 0, 0, 0	
					3D	3E	02	0286C	.ASCII	<2>\>=\	
					00	00	03	0286F	P.ATJ: .BYTE	3, 0	
						000D	02871	.WORD	13		
00	00	00	00	00	00	32	32	02873	.BYTE	50, 50, 0, 0, 0, 0, 0, 0	
						3D	01	0287B	.ASCII	<1>\>=\	
					00	00	03	0287D	P.ATK: .BYTE	3, 0	
						000E	0287F	.WORD	14		
00	00	00	00	00	00	32	32	02881	.BYTE	50, 50, 0, 0, 0, 0, 0, 0	
					3E	3C	02	02889	.ASCII	<2>\<>\	
						00000016	0288C	.LONG	22		
00002717	00002709	000026FB	000026ED	000026DF	000026D1	02890	P.ASO: .LONG	9937, 9951, 9965, 9979, 9993, 10007, -			
0000276B	0000275D	0000274F	00002741	00002733	00002725	028AB		10021, 10035, 10049, 10063, 10077, 10091, -			
000027C0	000027B2	000027A4	00002796	00002788	0000277A	028C0		10106, 10120, 10134, 10148, 10162, 10176, -			
		000027FA	000027EC	000027DD	000027CF	028DB	P.ATH: .BYTE	0, 0			
					00	00	028E8	.WORD	2		
						0002	028EA	.BYTE	0, 0, 0, 0		
			00	00	00	00	028EC	.ASCII	<1>\>\		
					5D	01	028F0	P.ATN: .BYTE	0, 2		
					02	00	028F2	.WORD	3		
						0003	028F4	.BYTE	0, 0, 0, 0		
			00	00	00	00	028F6	.ASCII	<1>\>:\		
					3A	01	028FA	P.ATO: .BYTE	0, 0		
					00	00	028FC	.WORD	1		
						0001	028FE	.BYTE	0, 0, 0, 0		
			00	00	00	00	02900	.ASCII	<1>\>:\		
					2C	01	02904	.BLKB	2		
						00000003	02906	.LONG	3		
						00000003	02908	P.ATL: .LONG	10341, 10351, 10361		
00002879	0000286F	00002865	00002861	00002857	00002853	0290C	P.ATQ: .BYTE	1, 40, 2, 0			
						00	02	02918	.LONG	1	
						00000001	0291C	.ASCII	<4>\TRUE\		
			45	55	52	54	04	02920	P.ATR: .BYTE	1, 40, 2, 0	
					00	02	28	01	02925	.LONG	0
						00000000	02929	.ASCII	<5>\FALSE\		
			45	53	4C	41	46	05	0292D	P.ATS: .BYTE	1, 0, 6, 0
					00	06	00	01	02933	.LONG	0
						00000000	02937	.ASCII	<3>\NIL\		
					4C	49	4E	03	0293B	.BLKB	1
						00000003	0293F	.LONG	3		
00002880	000028A2	00002895	0000288B	00002879	00002865	02940	P.ATP: .LONG	10389, 10402, 10416			
						00000000	02944	.LONG	0		
							02950	P.ATT: .BLKB	0		
						02	02954	P.ATU: .BYTE	2		
						01	02955	.BYTE	1		
						0009	02956	.WORD	9		
						03	02958	.BYTE	3		
						03	02959	.BYTE	3		

000B	0295A	.WORD	11
09	0295C	.BYTE	9
0F	0295D	.BYTE	15
000B	0295E	.WORD	11
05	02960	.BYTE	5
04	02961	.BYTE	4
0012	02962	.WORD	18
04	02964	.BYTE	4
07	02965	.BYTE	7
0017	02966	.WORD	23
07	02968	.BYTE	7
0A	02969	.BYTE	10
001C	0296A	.WORD	28
0A	0296C	.BYTE	10
0D	0296D	.BYTE	13
0021	0296E	.WORD	33
01	02970	.BYTE	1
0C	02971	.BYTE	12
0021	02972	.WORD	33
00000000	02974	.LONG	0
01	02978	.BYTE	1
02	02979	.BYTE	2
0021	0297A	.WORD	33
00000000	0297C	.LONG	0
03	02980	.BYTE	3
0E	02981	.BYTE	14
000B	02982	.WORD	11
09	02984	.BYTE	9
0F	02985	.BYTE	15
000B	02986	.WORD	11
05	02988	.BYTE	5
10	02989	.BYTE	16
0012	0298A	.WORD	18
04	0298C	.BYTE	4
12	0298D	.BYTE	18
0017	0298E	.WORD	23
07	02990	.BYTE	7
15	02991	.BYTE	21
001C	02992	.WORD	28
01	02994	.BYTE	1
17	02995	.BYTE	23
0021	02996	.WORD	33
00000000	02998	.LONG	0
05	0299C	.BYTE	5
19	0299D	.BYTE	25
0012	0299E	.WORD	18
04	029A0	.BYTE	4
1C	029A1	.BYTE	28
0017	029A2	.WORD	23
07	029A4	.BYTE	7
1F	029A5	.BYTE	31
001C	029A6	.WORD	28
01	029A8	.BYTE	1
21	029A9	.BYTE	33
0021	029AA	.WORD	33
00000000	029AC	.LONG	0
05	029B0	.BYTE	5

.....

					24	029B1	.BYTE	36	
					0012	029B2	.WORD	18	
					04	029B4	.BYTE	4	
					26	029B5	.BYTE	38	
					0017	029B6	.WORD	23	
					07	029B8	.BYTE	7	
					28	029B9	.BYTE	40	
					001C	029BA	.WORD	28	
					01	029BC	.BYTE	1	
					2A	029BD	.BYTE	42	
					0021	029BE	.WORD	33	
					00000000	029C0	.LONG	0	
					05	029C4	.BYTE	5	
					2C	029C5	.BYTE	44	
					0012	029C6	.WORD	18	
					04	029C8	.BYTE	4	
					2D	029C9	.BYTE	45	
					0017	029CA	.WORD	23	
					07	029CC	.BYTE	7	
					2E	029CD	.BYTE	46	
					001C	029CE	.WORD	28	
					01	029D0	.BYTE	1	
					2F	029D1	.BYTE	47	
					0021	029D2	.WORD	33	
					00000000	029D4	.LONG	0	
					00000000	029D8	.LONG	0	
00002889	000028D1	00000C8D	0000280D	000026B5	00002625	029DC	P.ATV: .LONG	9765, 9909, 10253, 3213, 10449, 10377, -	
00000000	00000000	00000001	00000001	000028D1	000028C1	029F4		10433, 10449, 1, 1, 0, 0, 0	
					00000000	02A0C			
					0000000B	02A10	.LONG	11	
21002800	7C002800	26000800	25000002	5F000007	24000007	02A14	P.ATW: .LONG	603979783, 1593835527, 620756994, -	
	3E042800	3C00A800	2D088850	5E012800	3D042800	02A2C		637536256, 2080385024, 553658368, -	
								1023682560, 1577134080, 755533904, -	
								1006675968, 1040459776	
					00000000	02A40	.LONG	0	
						02A44	P.ATX: .BLKB	0	
					01 02	02A44	P.ATZ: .BYTE	2, 1	
					0002	02A46	.WORD	2	
					00 00	02A48	.BYTE	0, 0, 0, 0, 0, 0, 0, 0	
					5C 01	02A50	.ASCII	<f><92>	
					01 03	02A52	P.AUA: .BYTE	3, 1	
					0003	02A54	.WORD	3	
					00 00	02A56	.BYTE	0, 0, 0, 0, 0, 0, 0, 0	
					5C 01	02A5E	.ASCII	<f><92>	
					01 04	02A60	P.AUB: .BYTE	4, 1	
					0004	02A62	.WORD	4	
					00 00	02A64	.BYTE	0, 0, 0, 0, 0, 0, 0, 0	
					28 01	02A6C	.ASCII	<f>\i\	
					01 03	02A6E	P.AUC: .BYTE	3, 1	
					0005	02A70	.WORD	5	
					00 00	02A72	.BYTE	0, 0, 0, 0, 0, 0, 0, 0	
					2E 01	02A7A	.ASCII	<f>\. \	
					02 02	02A7C	P.AUD: .BYTE	2, 2	
					000B	02A7E	.WORD	11	
					00 00	02A80	.BYTE	5, -56, 0, 0, 0, 0, 0, 0	
					28 01	02A88	.ASCII	<f>\i\	
					02 04	02A8A	P.AUE: .BYTE	4, 2	

00	00	00	00	00	00	06	00	02A8C	.WORD	12
						29	01	02A8E	.BYTE	-56, 6, 0, 0, 0, 0, 0, 0
						00	02	02A96	.ASCII	<1>\)\
								02A98	P.AUF: .BYTE	2, 0
								02A9A	.WORD	4
						C8	46	02A9C	.BYTE	70, -56, 0, 0, 0, 0, 0, 0
						2B	01	02AA4	.ASCII	<1>\+\
						00	02	02AA6	P.AUG: .BYTE	2, 0
								02AA8	.WORD	5
						C8	46	02AAA	.BYTE	70, -56, 0, 0, 0, 0, 0, 0
						2D	01	02AB2	.ASCII	<1>\-\
						00	02	02AB4	P.AUH: .BYTE	2, 0
								02AB6	.WORD	30
						C8	1E	02AB8	.BYTE	30, -56, 0, 0, 0, 0, 0, 0
						5E	01	02AC0	.ASCII	<1>\^\\
						00	03	02AC2	P.AUI: .BYTE	3, 0
								02AC4	.WORD	10
						5C	5A	02AC6	.BYTE	90, 92, 0, 0, 0, 0, 0, 0
						2A	02	02ACE	.ASCII	<2>*=\
						00	03	02AD1	P.AUJ: .BYTE	3, 0
								02AD3	.WORD	8
						50	50	02AD5	.BYTE	80, 80, 0, 0, 0, 0, 0, 0
						2A	01	02ADD	.ASCII	<1>*\
						00	03	02ADF	P.AUK: .BYTE	3, 0
								02AE1	.WORD	9
						50	50	02AE3	.BYTE	80, 80, 0, 0, 0, 0, 0, 0
						2F	01	02AEB	.ASCII	<1>\/\
						00	03	02AED	P.AUL: .BYTE	3, 0
								02AEF	.WORD	6
						3C	3C	02AF1	.BYTE	60, 60, 0, 0, 0, 0, 0, 0
						2B	01	02AF9	.ASCII	<1>\+\
						00	03	02AFB	P.AUM: .BYTE	3, 0
								02AFD	.WORD	7
						3C	3C	02AFF	.BYTE	60, 60, 0, 0, 0, 0, 0, 0
						2D	01	02B07	.ASCII	<1>\-\
						00	03	02B09	P.AUN: .BYTE	3, 0
								02B0B	.WORD	35
						37	37	02B0D	.BYTE	55, 55, 0, 0, 0, 0, 0, 0
						7C	02	02B15	.ASCII	<2>\!:\
						00	03	02B18	P.AUD: .BYTE	3, 0
								02B1A	.WORD	15
						32	32	02B1C	.BYTE	50, 50, 0, 0, 0, 0, 0, 0
						3E	01	02B24	.ASCII	<1>\>\
						00	03	02B26	P.AUP: .BYTE	3, 0
								02B28	.WORD	16
						32	32	02B2A	.BYTE	50, 50, 0, 0, 0, 0, 0, 0
						3C	01	02B32	.ASCII	<1>\<\
						00	03	02B34	P.AUQ: .BYTE	3, 0
								02B36	.WORD	21
						32	32	02B38	.BYTE	50, 50, 0, 0, 0, 0, 0, 0
						5E	02	02B40	.ASCII	<2>\^>\
						00	03	02B43	P.AUR: .BYTE	3, 0
								02B45	.WORD	17
						32	32	02B47	.BYTE	50, 50, 0, 0, 0, 0, 0, 0
		</								

00	00	00	00	00	00	32	32	02B56	.BYTE	50, 50, 0, 0, 0, 0, 0, 0
						3D	01	02B5E	.ASCII	<1>\=\
						00	03	02B60	.BYTE	3, 0
						000E		02B62	.WORD	14
00	00	00	00	00	00	32	32	02B64	.BYTE	50, 50, 0, 0, 0, 0, 0, 0
						3D	02	02B6C	.ASCII	<2>\^=\
						00	03	02B6F	.BYTE	3, 0
						0015		02B71	.WORD	24
00	00	00	00	00	00	32	32	02B73	.BYTE	50, 50, 0, 0, 0, 0, 0, 0
						3D	02	02B7B	.ASCII	<2>\<=\
						00	03	02B7E	.BYTE	3, 0
						0011		02B80	.WORD	17
00	00	00	00	00	00	32	32	02B82	.BYTE	50, 50, 0, 0, 0, 0, 0, 0
						3D	02	02B8A	.ASCII	<2>\>=\
						00	03	02B8D	.BYTE	3, 0
						001F		02B8F	.WORD	31
00	00	00	00	00	00	2D	2D	02B91	.BYTE	45, 45, 0, 0, 0, 0, 0, 0
						26	01	02B99	.ASCII	<1>\8\
						00	03	02B9B	.BYTE	3, 0
						0020		02B9D	.WORD	32
00	00	00	00	00	00	28	28	02B9F	.BYTE	40, 40, 0, 0, 0, 0, 0, 0
						7C	01	02BA7	.ASCII	<1>\!\
								02BA9	.BLKB	3
								02BAC	.LONG	25
00002A07	000029F9	000029EB	000029DD	000029CF	00000019	000029C1	02B80	P.ATY:	.LONG	10689, 10703, 10717, 10731, 10745, 10759, -
00002A5C	00002A4E	00002A3F	00002A31	00002A23	000029C1	00002A15	02BC8			10773, 10787, 10801, 10815, 10830, 10844, -
00002AB1	00002AA3	00002A95	00002A86	00002A78	00002A6A	00002A6A	02BE0			10858, 10872, 10886, 10901, 10915, 10929, -
00002B0A	00002AFB	00002AEC	00002ADD	00002ACF	00002AC0	00002AC0	02BF8			10944, 10959, 10973, 10988, 11003, 11018, -
					00002B18	00002B18	02C10			11032
					01	03	02C14	P.AUY:	.BYTE	3, 1
					0008		02C16		.WORD	8
00	00	00	00	00	00	00	00	02C18	.BYTE	0, 0, 0, 0, 0, 0, 0, 0
					3E	2D	02	02C20	.ASCII	<2>\->\
						00	00	02C23	.BYTE	0, 0
					0002		02C25	P.AVA:	.WORD	2
					00	00	00	02C27	.BYTE	0, 0, 0, 0
					29	01	02C2B		.ASCII	<1>\)\
					02	00	02C2D	P.AVB:	.BYTE	0, 2
					0003		02C2F		.WORD	3
					00	00	00	02C31	.BYTE	0, 0, 0, 0
					3A	01	02C35		.ASCII	<1>\:\
					00	00	02C37	P.AVC:	.BYTE	0, 0
					0001		02C39		.WORD	1
					00	00	00	02C3B	.BYTE	0, 0, 0, 0
					2C	01	02C3F		.ASCII	<1>\;\
							02C41		.BLKB	3
					00000003		02C44		.LONG	3
00002BB4	00002BAA				00002BA0	00002BA0	02C48	P.AUZ:	.LONG	11168, 11178, 11188
					00000000	00000000	02C54		.LONG	0
					00000000		02C58	P.AVD:	.BLKB	0
							02C58		.LONG	0
							02C5C	P.AVE:	.BLKB	0
					01		02C5C	P.AVF:	.BYTE	1
					0E		02C5D		.BYTE	14
					0005		02C5E		.WORD	3
					03		02C60		.BYTE	3
					01		02C61		.BYTE	1

0003	02C62	.WORD	3
00	02C64	.BYTE	0
0A	02C65	.BYTE	10
0021	02C66	.WORD	33
01	02C68	.BYTE	1
0F	02C69	.BYTE	15
0012	02C6A	.WORD	18
00	02C6C	.BYTE	0
0A	02C6D	.BYTE	10
0021	02C6E	.WORD	33
01	02C70	.BYTE	1
0E	02C71	.BYTE	14
0005	02C72	.WORD	5
03	02C74	.BYTE	3
01	02C75	.BYTE	1
0012	02C76	.WORD	18
02	02C78	.BYTE	2
01	02C79	.BYTE	1
000C	02C7A	.WORD	12
06	02C7C	.BYTE	6
01	02C7D	.BYTE	1
000C	02C7E	.WORD	12
07	02C80	.BYTE	7
01	02C81	.BYTE	1
000C	02C82	.WORD	12
08	02C84	.BYTE	8
01	02C85	.BYTE	1
000C	02C86	.WORD	12
00	02C88	.BYTE	0
10	02C89	.BYTE	16
0021	02C8A	.WORD	33
01	02C8C	.BYTE	1
01	02C8D	.BYTE	1
000C	02C8E	.WORD	12
02	02C90	.BYTE	2
01	02C91	.BYTE	1
000C	02C92	.WORD	12
06	02C94	.BYTE	6
01	02C95	.BYTE	1
000C	02C96	.WORD	12
07	02C98	.BYTE	7
01	02C99	.BYTE	1
000C	02C9A	.WORD	12
08	02C9C	.BYTE	8
01	02C9D	.BYTE	1
000C	02C9E	.WORD	12
00	02CA0	.BYTE	0
09	02CA1	.BYTE	9
0021	02CA2	.WORD	33
01	02CA4	.BYTE	1
0F	02CA5	.BYTE	15
0012	02CA6	.WORD	18
03	02CA8	.BYTE	3
08	02CA9	.BYTE	8
0021	02CAA	.WORD	33
08	02CAC	.BYTE	8
05	02CAD	.BYTE	5

.....

0019	02CAE	.WORD	25
07	02CB0	.BYTE	7
06	02CB1	.BYTE	6
0019	02CB2	.WORD	25
0D	02CB4	.BYTE	13
12	02CB5	.BYTE	18
0019	02CB6	.WORD	25
09	02CB8	.BYTE	9
07	02CB9	.BYTE	7
0019	02CBA	.WORD	25
00	02CBC	.BYTE	0
10	02CBD	.BYTE	16
0021	02CBE	.WORD	33
01	02CC0	.BYTE	1
01	02CC1	.BYTE	1
001F	02CC2	.WORD	31
04	02CC4	.BYTE	4
01	02CC5	.BYTE	1
001D	02CC6	.WORD	29
05	02CC8	.BYTE	5
01	02CC9	.BYTE	1
001D	02CCA	.WORD	29
00	02CCC	.BYTE	0
08	02CCD	.BYTE	8
0021	02CCE	.WORD	33
01	02CD0	.BYTE	1
01	02CD1	.BYTE	1
001F	02CD2	.WORD	31
00	02CD4	.BYTE	0
09	02CD5	.BYTE	9
0021	02CD6	.WORD	33
01	02CD8	.BYTE	1
01	02CD9	.BYTE	1
001F	02CDA	.WORD	31
00	02CDC	.BYTE	0
09	02CDD	.BYTE	9
0021	02CDE	.WORD	33
00	02CE0	.BYTE	0
0B	02CE1	.BYTE	11
0021	02CE2	.WORD	33
02	02CE4	.BYTE	2
01	02CE5	.BYTE	1
0008	02CE6	.WORD	8
03	02CE8	.BYTE	3
03	02CE9	.BYTE	3
000A	02CEA	.WORD	10
09	02CEC	.BYTE	9
0F	02CED	.BYTE	15
000A	02CEE	.WORD	10
05	02CF0	.BYTE	5
05	02CF1	.BYTE	5
0011	02CF2	.WORD	17
04	02CF4	.BYTE	4
09	02CF5	.BYTE	9
0016	02CF6	.WORD	22
08	02CF8	.BYTE	8
0B	02CF9	.BYTE	11

P.AVG:

.....

0000	02CFA	.WORD	0
01	02CFC	.BYTE	1
0C	02CFD	.BYTE	12
001B	02CFE	.WORD	27
00000000	02D00	.LONG	0
01	02D04	.BYTE	1
02	02D05	.BYTE	2
001B	02D06	.WORD	27
00000000	02D08	.LONG	0
03	02D0C	.BYTE	3
0E	02D0D	.BYTE	14
000A	02D0E	.WORD	10
09	02D10	.BYTE	9
0F	02D11	.BYTE	15
000A	02D12	.WORD	10
05	02D14	.BYTE	5
11	02D15	.BYTE	17
0011	02D16	.WORD	17
04	02D18	.BYTE	4
13	02D19	.BYTE	19
0016	02D1A	.WORD	22
08	02D1C	.BYTE	8
16	02D1D	.BYTE	22
0000	02D1E	.WORD	0
01	02D20	.BYTE	1
17	02D21	.BYTE	23
001B	02D22	.WORD	27
00000000	02D24	.LONG	0
05	02D28	.BYTE	5
1A	02D29	.BYTE	26
0011	02D2A	.WORD	17
04	02D2C	.BYTE	4
1D	02D2D	.BYTE	29
0016	02D2E	.WORD	22
08	02D30	.BYTE	8
20	02D31	.BYTE	32
0000	02D32	.WORD	0
01	02D34	.BYTE	1
22	02D35	.BYTE	34
001B	02D36	.WORD	27
00000000	02D38	.LONG	0
05	02D3C	.BYTE	5
25	02D3D	.BYTE	37
0011	02D3E	.WORD	17
04	02D40	.BYTE	4
27	02D41	.BYTE	39
0016	02D42	.WORD	22
08	02D44	.BYTE	8
29	02D45	.BYTE	41
0000	02D46	.WORD	0
01	02D48	.BYTE	1
2B	02D49	.BYTE	43
001B	02D4A	.WORD	27
00000000	02D4C	.LONG	0
00000000	02D50	.LONG	0
00002991	02D54	.LONG	10641, 10689, 11053, 11225, 11361, 11205, -
00002BD5	02D6C		11221, 11225, 0, 1, 0, 1, 0

000028C5 00002C61 00002BD9 00002B2D 000029C1
00000001 00000000 00000001 00000000 00002BD9

P.AVH:

10641, 10689, 11053, 11225, 11361, 11205, -
11221, 11225, 0, 1, 0, 1, 0

```
5F000007 23000007 3D042800 3E042800 3C04A800 00000000 02D84
00000007 02D88
2A000000 02D8C P.AVI: .LONG 7
24000007 02DA4 704643072, 1006938112, 1040459776, -
1023682560, 587202567, 1593835527, -
603979783
00 02 02DA8 P.AVK: .BYTE 2 0
0017 02DAA .WORD 23
00 00 00 00 00 00 C8 0B 02DAC .BYTE 11, -56, 0, 0, 0, 0, 0, 0
54 4F 4E 03 02DB4 .ASCII <3>\NOT\
00 03 02DB8 P.AVL: .BYTE 3 0
0018 02DBA .WORD 24
00 00 00 00 00 00 0A 0A 02DBC .BYTE 10, 10, 0, 0, 0, 0, 0, 0
44 4E 41 03 02DC4 .ASCII <3>\AND\
00 03 02DC8 P.AVM: .BYTE 3 0
0019 02DCA .WORD 25
00 00 00 00 00 00 0A 0A 02DCC .BYTE 10, 10, 0, 0, 0, 0, 0, 0
52 4F 02 02DD4 .ASCII <2>\OR\
02 03 02DD7 P.AVN: .BYTE 3 2
002C 02DD9 .WORD 44
00 00 00 00 00 00 0F 0F 02DDB .BYTE 15, 15, 0, 0, 0, 0, 0, 0
54 4F 4E 03 02DE3 .ASCII <3>\NOT\
02DE7 .BLKB 1
00000004 02DE8 .LONG 4
00002D54 00002D45 00002D35 00002D25 02DEC P.AVJ: .LONG 11557, 11573, 11589, 11604
00 03 02DFC P.AVO: .BYTE 3 0
000E 02DFE .WORD 14
00 00 00 00 00 00 0F 0F 02E00 .BYTE 15, 15, 0, 0, 0, 0, 0, 0
3D 20 54 4F 4E 05 02E08 .ASCII <5>\NOT =\
00 03 02E0E P.AVP: .BYTE 3 0
0015 02E10 .WORD 21
00 00 00 00 00 00 0F 0F 02E12 .BYTE 15, 15, 0, 0, 0, 0, 0, 0
3E 20 54 4F 4E 05 02E1A .ASCII <5>\NOT >\
00 03 02E20 P.AVQ: .BYTE 3 0
0011 02E22 .WORD 17
00 00 00 00 00 00 0F 0F 02E24 .BYTE 15, 15, 0, 0, 0, 0, 0, 0
3C 20 54 4F 4E 05 02E2C .ASCII <5>\NOT <\
01 02 02E32 P.AVS: .BYTE 2 1
0002 02E34 .WORD 2
00 00 00 00 00 00 00 00 02E36 .BYTF 0 0, 0, 0, 0, 0, 0, 0, 0
5C 01 02E3E .ASCII <1><92>
01 03 02E40 P.AVT: .BYTE 3 1
0003 02E42 .WORD 3
00 00 00 00 00 00 00 00 02E44 .BYTE 0 0, 0, 0, 0, 0, 0, 0, 0
5C 01 02E4C .ASCII <1><92>
01 04 02E4E P.AVU: .BYTE 4 1
0004 02E50 .WORD 4
00 00 00 00 00 00 00 00 02E52 .BYTE 0 0, 0, 0, 0, 0, 0, 0, 0
28 01 02E5A .ASCII <1>\i\
02 02 02E5C P.AVV: .BYTE 2 2
003F 02E5E .WORD 64
00 00 00 00 00 00 C8 0F 02E60 .BYTE 15, -56, 0, 0, 0, 0, 0, 0
3E 01 02E68 .ASCII <1>\>\
02 02 02E6A P.AVW: .BYTE 2 2
0040 02E6C .WORD 64
00 00 00 00 00 00 C8 0F 02E6E .BYTE 15, -56, 0, 0, 0, 0, 0, 0
3C 01 02E76 .ASCII <1>\<\
02 02 02E78 P.AVX: .BYTE 2 2
```


[illegible]

			0003	02F72	.WORD	3		
	00	00	00	02F74	.BYTE	0	0, 0, 0	
			3A	01	02F78	.ASCII	<f>\:\	
			00	00	02F7A	.BYTE	0, 0	
			0001	02F7C	.WORD	1		
	00	00	00	02F7E	.BYTE	0	0, 0, 0	
			2C	01	02F82	.ASCII	<f>\:\	
00002EF7	00002EED		00000003	02F84	.LONG	3		
			00002EE3	02F88	P.AWJ: .LONG	12003, 12013, 12023		
			00000000	02F94	.LONG	0		
			00000000	02F98	P.AWN: .BLKB	0		
				02F98	.LONG	0		
			02	02F9C	P.AWO: .BLKB	0		
			01	02F9C	P.AWP: .BYTE	2		
				02F9D	.BYTE	1		
			0006	02F9E	.WORD	6		
			03	02FA0	.BYTE	3		
			03	02FA1	.BYTE	3		
			0008	02FA2	.WORD	8		
			09	02FA4	.BYTE	9		
			0F	02FA5	.BYTE	15		
			0008	02FA6	.WORD	8		
			04	02FA8	.BYTE	4		
			07	02FA9	.BYTE	7		
			000D	02FAA	.WORD	13		
			01	02FAC	.BYTE	1		
			0C	02FAD	.BYTE	12		
			0010	02FAE	.WORD	16		
			00000000	02FB0	.LONG	0		
			01	02FB4	.BYTE	1		
			02	02FB5	.BYTE	2		
			0010	02FB6	.WORD	16		
			00000000	02FB8	.LONG	0		
			03	02FBC	.BYTE	3		
			0E	02FBD	.BYTE	14		
			0008	02FBE	.WORD	8		
			09	02FC0	.BYTE	9		
			0F	02FC1	.BYTE	15		
			0008	02FC2	.WORD	8		
			04	02FC4	.BYTE	4		
			12	02FC5	.BYTE	18		
			000D	02FC6	.WORD	13		
			01	02FC8	.BYTE	1		
			17	02FC9	.BYTE	23		
			0010	02FCA	.WORD	16		
			00000000	02FCC	.LONG	0		
			04	02FD0	.BYTE	4		
			26	02FD1	.BYTE	38		
			000D	02FD2	.WORD	13		
			01	02FD4	.BYTE	1		
			2A	02FD5	.BYTE	42		
			0010	02FD6	.WORD	16		
			00000000	02FD8	.LONG	0		
			00000000	02FDC	.LONG	0		
00002F05	00002F19	00002BD9	00002E95	00002D69	00002D09	P.AWQ: .LONG	11529, 11625, 11925, 11225, 12057, 12037, -	
00000000	00000000	00000001	00000000	00002BD1	00002F15		12053, 10449, 0, 1, 0, 0, 0	
			00000000	03010				

00002CD1 0000153D 00001FC9 000018F9 000022F1 000025ED 03014 LANGUAGE_TABLE_PTRS:
00000DA1 00001229 00002F5D 00001C45 00002959 0302C .LONG 9709, 8945, 6393, 8137, 5437, 11473, -
10585, 7237, 12125, 4649, 3489

.PSECT DBG\$OWN,NOEXE, PIC,2

00000 ADDRESS_LENGTH:
.BLKB 4
00004 ADDRESS_TYPE:
.BLKB 4
00008 BIF_TABLE:
.BLKB 4
0000C CASING_SIGNIFICANT:
.BLKB 4
00010 CHARPTR:.BLKB 4
00014 CHARTBL:.BLKB 1024
00414 COMPONENTS_IN_PATHNAME:
.BLKB 4
00418 ENFORCE_RECORD:
.BLKB 4
0041C EXPRESSION_RADIX:
.BLKB 4
00420 IDENT_OPERATOR_TABLE:
.BLKB 4
00424 INCOMPLETE_QUAL:
.BLKB 4
00428 MULTIPLE_SUBSCR:
.BLKB 4
0042C OPCHAR_OPERATOR_TABLE:
.BLKB 4
00430 PRIMARY_TABLE:
.BLKB 4
00000000 00434 SAVED_TOKEN:
.LONG 0
00438 STATE_TABLE:
.BLKB 4
0043C SUBSCRIPT_TERM_TBL:
.BLKB 4
00440 PRIDTBL:.BLKB 4
00000000 00444 TERMINATOR_CODE:
.LONG 0
00000000 00448 TERMINATOR_LENGTH:
.LONG 0
0044C VARSTACK1:
.BLKB 80
0049C VARSTACK2:
.BLKB 80
004EC VARSTACK3:
.BLKB 80
0053C VARSTK_INDEX:
.BLKB 4

.PSECT DBG\$GLOBAL,NOEXE, PIC,2

00000 DBG\$GL_CHARTBL:.
.BLKB 4

```

DBG$GL_CONVERT_TOKEN==
DBG$GL_DEPOSIT_TOKEN==
DBG$GL_IDENTITY_TOKEN==
DBG$GL_NEG_CONST_TOKEN==
DBG$GL_POS_CONST_TOKEN==
DBG$GL_NEG_SIGN_TOKEN==
DBG$GL_POS_SIGN_TOKEN==
TABLEBASE=
PERCENT_TABLE=
INITIATOR_TOKEN=
TERMINATOR_TOKEN=
PRIMARY_TERM_TOKEN=
RADIX_OP_DEC=
RADIX_OP_HEX=
RADIX_OP_OCT=
RADIX_OP_BIN=
CURLOC_TOKEN=
CURVAL_TOKEN=
PREVLOC_TOKEN=
ADDR_EXPR_OPTBL=
EMPTY_TERM_TBL=
COMMA_TERM_TBL=
EQUAL_TERM_TBL=
DO_TERM_TBL=
THEN_TERM_TBL=
CONCOL_TERM_TBL=
CMWHD0_TERM_TBL=
OPEN_TERM_TBL=
COMPAREN_TERM_TBL=
TO_TERM_TBL=
BY_TERM_TBL=
BIT_SELECT_TERM_TBL=
SET_CONSTANT_TERM_TBL=
UNKNOWN_CHARTBL=
UNKNOWN_IDENT_OPTBL=
UNKNOWN_OPCHAR_OPTBL=
UNKNOWN_SUBSCR_TERM_TBL=
UNKNOWN_NUMBER_TABLE=
UNKNOWN_PRID_TABLE=
UNKNOWN_FUNCTION_TABLE=
UNKNOWN_PRIMARY_TABLE=
UNKNOWN_TABLES=
ADA_CHARTBL=
ADA_IDENT_OPTBL=

```



```

ADA_OPCHAR_OPTBL= P.AGY
ADA_TICK_TOKEN= P.AHT
ADA_SUBSCR_TERM_TBL=P.AID
ADA_PRID_TABLE= P.AII
ADA_FUNCTION_TABLE= P.AIJ
ADA_NUMBER_TABLE= P.AIK
ADA_PRIMARY_TABLE= P.AIL
ADA_TABLES= P.AIM
BASIC_CHARTBL= P.AIN
BASIC_IDENT_OPTBL= P.AIO
BASIC_OPCHAR_OPTBL= P.AIV
BASIC_SUBSCR_TERM_TBL=
P.AJT
BASIC_PRID_TABLE= P.AJX
BASIC_FUNCTION_TABLE=
P.AJY
BASIC_NUMBER_TABLE= P.AGK
BASIC_PRIMARY_TABLE=P.AJZ
BASIC_TABLES= P.AKA
BLISS_CHARTBL= P.AKB
BLISS_IDENT_OPTBL= P.AKC
BLISS_OPCHAR_OPTBL= P.ALB
BLISS_SUBSCR_TERM_TBL=
P.ALQ
BLISS_PRID_TABLE= P.ALU
BLISS_FUNCTION_TABLE=
P.ALV
BLISS_NUMBER_TABLE= P.AGK
BLISS_PRIMARY_TABLE=P.ALW
BLISS_TABLES= P.ALX
C_CHARTBL= P.ALY
C_IDENT_OPTBL= P.ALZ
C_OPCHAR_OPTBL= P.AMB
C_ADDR_OF_TOKEN= P.AMZ
C_BIT_AND_TOKEN= P.ANA
C_AND_TOKEN= P.ANB
C_ADD_TOKEN= P.ANC
C_MINUS_TOKEN= P.AND
C_SUB_TOKEN= P.ANE
C_ARROW_TOKEN= P.ANF
C_INDIRECT_TOKEN= P.ANG
C_SUBSCR_TERM_TBL= P.ANH
C_PRID_TABLE= P.ANL
C_FUNCTION_TABLE= P.ANM
C_NUMBER_TABLE= P.AGK
C_PRIMARY_TABLE= P.ANN
C_TABLES= P.ANO
COBOL_CHARTBL= P.ANP
COBOL_IDENT_OPTBL= P.ANQ
COBOL_NOT_EQ_TOKEN=P.ANX
COBOL_NOT_GTR_TOKEN=P.ANY
COBOL_NOT_LSS_TOKEN=P.ANZ
COBOL_OPCHAR_OPTBL= P.AOA
COBOL_SUBSCR_TERM_TBL=
P.AOS
COBOL_PRID_TABLE= P.AOW
COBOL_FUNCTION_TABLE=

```

```

COBOL_NUMBER_TABLE= P.AOX
COBOL_PRIMARY_TABLE= P.AOY
COBOL_TABLES= P.AOZ
FORTRAN_CHARTBL= P.APA
FORTRAN_IDENT_OPTBL= P.APB
FORTRAN_OPCHAR_OPTBL= P.APC
FORTRAN_INDIRECT_TOKEN= P.APD
FORTRAN_DOT_TOKEN= P.APS
FORTRAN_SPECIAL_OPTBL= P.APT
FORTRAN_SUBSCR_TERM_TBL= P.APU
FORTRAN_PRID_TABLE= P.AQH
FORTRAN_FUNCTION_TABLE= P.AQL
FORTRAN_NUMBER_TABLE= P.AQO
FORTRAN_PRIMARY_TABLE= P.AGK
FORTRAN_TABLES= P.AQP
MACRO_CHARTBL= P.AQQ
MACRO_IDENT_OPTBL= P.AQR
MACRO_OPCHAR_OPTBL= P.AQS
MACRO_SUBSCR_TERM_TBL= P.ARL
MACRO_PRID_TABLE= P.ASA
MACRO_FUNCTION_TABLE= P.ASB
MACRO_NUMBER_TABLE= P.ASC
MACRO_PRIMARY_TABLE= P.ASD
MACRO_TABLES= P.ASE
PASCAL_CHARTBL= P.ASF
PASCAL_IDENT_OPTBL= P.ASG
PASCAL_OPCHAR_OPTBL= P.ASO
PASCAL_SUBSCR_TERM_TBL= P.ATL
PASCAL_PRID_TABLE= P.ATP
PASCAL_FUNCTION_TABLE= P.ATT
PASCAL_NUMBER_TABLE= P.AGK
PASCAL_PRIMARY_TABLE= P.ATU
PASCAL_TABLES= P.ATV
PLI_CHARTBL= P.ATW
PLI_IDENT_OPTBL= P.ATX
PLI_OPCHAR_OPTBL= P.ATY
PLI_ARROW_TOKEN= P.AUY
PLI_SUBSCR_TERM_TBL= P.AUZ
PLI_PRID_TABLE= P.AVD
PLI_FUNCTION_TABLE= P.AVE
PLI_NUMBER_TABLE= P.AVF
PLI_PRIMARY_TABLE= P.AVG
PLI_TABLES= P.AVH
RPG_CHARTBL= P.AVI

```

```

RPG_IDENT_OPTBL= P.AVJ
RPG_NOT_EOL_TOKEN= P.AVO
RPG_NOT_GTR_TOKEN= P.AVP
RPG_NOT_LSS_TOKEN= P.AVQ
RPG_OPCRAR_OPTBL= P.AVR
RPG_MULTIPLY_TOKEN= P.AWI
RPG_SUBSCR_TERM_TBL=P.AWJ
RPG_PRID_TABLE= P.AWN
RPG_FUNCTION_TABLE= P.AWO
RPG_NUMBER_TABLE= P.AVF
RPG_PRIMARY_TABLE= P.AWP
RPG_TABLES= P.AWQ
.EXTRN DBG$DATA_LENGTH
.EXTRN DBG$DEF_SYM_FIND
.EXTRN DBG$DUMP_HEX, DBG$ENUM_POS
.EXTRN DBG$ENUM_VAL, DBG$EVAL_OP_SET_LANGUAGE
.EXTRN DBG$EVAL_ADA_TICK
.EXTRN DBG$EVAL_ADDR_OPERATOR
.EXTRN DBG$EVAL_LANG_OPERATOR
.EXTRN DBG$GET_TEMPMEM
.EXTRN DBG$HASH_FIND, DBG$HASH_FIND_SETUP
.EXTRN DBG$PERFORM_TYPEID_CHECK
.EXTRN DBG$MAKE_SKELETON_DESC
.EXTRN DBG$MAP_DTYPE_CLASS
.EXTRN DBG$NCOB_PATHDESC_TO_CS
.EXTRN DBG$NCOPY_DESC, DBG$NEWLINE
.EXTRN DBG$NPATHDESC_TO_CS
.EXTRN DBG$NUM_BYTES, DBG$PRIM_TO_ADDR
.EXTRN DBG$PRIM_TO_VAL
.EXTRN DBG$PRINT, DBG$PRINT_SET_LANGUAGE
.EXTRN DBG$STA_GETSYMBOL
.EXTRN DBG$STA_SETCONTEXT
.EXTRN DBG$STA_SYMSIZE
.EXTRN DBG$STA_SYMTYPE
.EXTRN DBG$STA_SYMVALUE
.EXTRN DBG$STA_TYPEFCODE
.EXTRN DBG$STA_TYP_ARRAY
.EXTRN DBG$STA_TYP_ATOMIC
.EXTRN DBG$STA_TYP_DESCR
.EXTRN DBG$STA_TYP_FILE
.EXTRN DBG$STA_TYP_RECORD
.EXTRN DBG$STA_TYP_SUBRNG
.EXTRN DBG$STA_TYP_TYPEDPTR
.EXTRN DBG$STA_TYP_VARIANT
.EXTRN DBG$STA_TYP_VARIANT_COMP
.EXTRN DBG$TYPEID_FOR_ARRAY
.EXTRN DBG$TYPEID_FOR_ATOMIC
.EXTRN DBG$TYPEID_FOR_SET
.EXTRN OT$SCVT_TIL, DBG$GL_ARRSUB_FLAG
.EXTRN DBG$GL_DEVELOPER
.EXTRN DBG$GL_CURRENT_PRIMARY
.EXTRN DBG$GB_LANGUAGE
.EXTRN DBG$GB_MOD_PTR, DBG$GL_ORIG_COMMAND_PTR
.EXTRN DBG$GL_RECMP_FLAG
.EXTRN DBG$GB_SET_BREAK_FLAG
.EXTRN DBG$GL_UPCASE_COMMAND_PTR

```

DBGPARSER
V04-000

1-7
16-Sep-1984 02:10:13 VAX-11 Bliss-32 V4.0-742
14-Sep-1984 12:17:30 [DEBUG.SRC]DBGPARSER.B32;1

Page 127
(17)

.PSECT DBG\$CODE,NOWRT, SHR, PIC,0

0000 00000 AAA_DUMMY:

04 00002

.WORD Save nothing
RET

: 3379
: 3397

; Routine Size: 3 bytes. Routine Base: DBG\$CODE + 0000


```
3282 3398 1 GLOBAL ROUTINE DBGSADDR_EXP_INT(INPUT_DESC, ADDR_EXP_PTR, TYPE, LENGTH,  
3283 3399 1 RADIX, TERM_INDEX) =  
3284 3400 1  
3285 3401 1 FUNCTION  
3286 3402 1 This is the Address Expression Interpreter for most languages  
3287 3403 1 supported by DEBUG. It parses and evaluates a DEBUG address  
3288 3404 1 expression and returns an Address Expression Descriptor which  
3289 3405 1 represents the value of the expression. This routine itself is  
3290 3406 1 only a set-up routine which sets up the character pointer and  
3291 3407 1 the expression radix to use and then calls DBG$EXPRESSION_PARSER  
3292 3408 1 to do the actual work.  
3293 3409 1  
3294 3410 1 INPUTS  
3295 3411 1 INPUT_DESC - A string descriptor which points to the input string to be  
3296 3412 1 parsed as an Address Expression. Only the pointer field of  
3297 3413 1 this descriptor is actually used--the string is expected to  
3298 3414 1 terminated by a carriage-return character.  
3299 3415 1  
3300 3416 1 ADDR_EXP_PTR - The address of a longword location to receive a pointer  
3301 3417 1 to a Primary or Value Descriptor returned by this routine.  
3302 3418 1  
3303 3419 1 TYPE - The address of a longword location to receive the address  
3304 3420 1 "type", namely instruction or not instruction.  
3305 3421 1  
3306 3422 1 LENGTH - The address of a longword location to receive the length of  
3307 3423 1 the current instruction if the "type" is instruction.  
3308 3424 1  
3309 3425 1 RADIX - The radix to be used to interpret integer numbers. The al-  
3310 3426 1 lowed radix values are DBG$K_DECIMAL, DBG$K_HEX, DBG$K_OCTAL,  
3311 3427 1 and DBG$K_BINARY.  
3312 3428 1  
3313 3429 1 TERM_INDEX - A "terminator index" which indicates which lexical tokens  
3314 3430 1 are allowed as expression terminators in this context. For  
3315 3431 1 example, in the EXAMINE command, "." and ":" are allowed  
3316 3432 1 terminators and in the DEPOSIT command, "=" is the allowed  
3317 3433 1 terminator. These index values have names of the form  
3318 3434 1 TOKEN$K_TERM_xxx.  
3319 3435 1  
3320 3436 1 OUTPUTS  
3321 3437 1 INPUT_DESC - The input string descriptor is updated to point to the  
3322 3438 1 first character after the address expression just parsed.  
3323 3439 1 If the parse was stopped by a terminator token, the input  
3324 3440 1 string descriptor will point to that token on return.  
3325 3441 1  
3326 3442 1 ADDR_EXP_PTR - A Primary or Value Descriptor is constructed and a  
3327 3443 1 pointer to that descriptor is returned to ADDR_EXP_PTR.  
3328 3444 1  
3329 3445 1 TYPE - If the address expression yields an instruction address, the  
3330 3446 1 value DBG$K_INSTRUCTION is returned to TYPE. Otherwise, the  
3331 3447 1 value DBG$K_NOTYPE is returned to TYPE.  
3332 3448 1  
3333 3449 1 LENGTH - If the value DBG$K_INSTRUCTION is returned to TYPE, the  
3334 3450 1 length in bytes of the instruction pointed to by the address  
3335 3451 1 expression is returned to LENGTH. Otherwise, 0 is returned.  
3336 3452 1  
3337 3453 1 The value ST$K_SUCCESS is return as the routine result if the expres-  
3338 3454 1 sion was terminated by a carriage-return character. If it
```

was terminated any other way (i.e., by a terminator token),
the value STS\$K_WARNING is returned.

BEGIN

MAP

INPUT_DESC: REF BLOCK[.BYTE], : Pointer to input string descriptor
ADDR_EXP_PTR: REF VECTOR[1], : Longword to receive address of a
: Primary or Value Descriptor
TYPE: REF VECTOR[1], : Longword to receive type code value
LENGTH: REF VECTOR[1]; : Longword to receive length value

: Set up CHARPTR to point to the start of the expression string. Also set
up the radix we are to use in the scan and initialize some variables.

CHARPTR = .INPUT_DESC[DSC\$A_POINTER];
EXPRESSION_RADIX = .RADIX;
SAVED_TOKEN = 0;

: Set the address "type" and "length" to indicate "no type". These values
may be changed to "instruction" if the Expression Parser finds that the
address expression points to an instruction address. (This allows DEBUG
to automatically display instruction locations as instructions.)

ADDRESS_TYPE = DBG\$K_NOTYPE;
ADDRESS_LENGTH = 0;

: Call the Expression Parser to parse the address expression. Then fix up
the string descriptor to reflect the new location of the parse pointer,
return the address type and length values, and return the appropriate
status code.

ADDR_EXP_PTR[0] = DBG\$EXPRESSION_PARSER(TRUE,
: TERM_POINTER_TBL[.TERM_INDEX] + TABLEBASE);
INPUT_DESC[DSC\$W_LENGTH] = .INPUT_DESC[DSC\$W_LENGTH] +
: .INPUT_DESC[DSC\$A_POINTER] - .CHARPTR;
INPUT_DESC[DSC\$A_POINTER] = .CHARPTR;
TYPE[0] = .ADDRESS_TYPE;
LENGTH[0] = .ADDRESS_LENGTH;
IF .CHARPTR[0] NEQ CAR_RET THEN RETURN STS\$K_WARNING;
RETURN STS\$K_SUCCESS;

END;

53 00000000' 000C 00000
52 04 EF 9E 00002
63 04 AC D0 00009
A2 D0 00000

.ENTRY DBG\$ADDR_EXP_INT, Save R2,R3
MOVAB CHARPTR, R3
MOVL INPUT_DESC, R2
MOVL 4(R2), CHARPTR

: 3398

: 3473

040C	C3	14	AC	D0	00011	MOVL	RADIX, EXPRESSION_RADIX	3474
		0424	C3	D4	00017	CLRL	SAVED_TOKEN	3475
F4	A3	80	8F	9A	00018	MOVZBL	#128, ADDRESS_TYPE	3483
		F0	A3	D4	00020	CLRL	ADDRESS_LENGTH	3484
	50	18	AC	D0	00023	MOVL	TERM_INDEX, R0	3493
	50	00000000	'EF40	D0	00027	MOVL	TERM_POINTER_TBL[R0], R0	
		00000000	'EF40	9F	0002F	PUSHAB	TABLEBASE[R0]	
			01	DD	00036	PUSHL	#1	3492
0000V	CF		02	FB	00038	CALLS	#2, DBG\$EXPRESSION_PARSER	
08	BC		50	D0	0003D	MOVL	R0, @ADDR_EXP_PTR	
	51		62	3C	00041	MOVZWL	(R2), R1	3495
	51	04	A2	C0	00044	ADDL2	4(R2), R1	
	50		63	D0	00048	MOVL	CHARPTR, R0	
62	51		50	A3	0004B	SUBW3	R0, R1, (R2)	
	04		50	D0	0004F	MOVL	R0, 4(R2)	3496
	0C	F4	A3	D0	00053	MOVL	ADDRESS_TYPE, @TYPE	3497
	10	F0	A3	D0	00058	MOVL	ADDRESS_LENGTH, @LENGTH	3498
			60	91	0005D	CMPB	(R0), #T3	3499
			04	12	00060	BNEQ	1\$	
	50		01	D0	00062	MOVL	#1, R0	3500
			04	04	00065	RET		
			50	D4	00066	CLRL	R0	3502
			04	04	00068	RET		

; Routine Size: 105 bytes. Routine Base: DBG\$CODE + 0003

```
3388 3503 1 GLOBAL ROUTINE DBGSBUILD_PRIMARY_SUBNODE(PRIMPTR, KIND, SYMID,
3389 3504 1 FCODE, TYPEID, DSTPTR): NOVALUE =
3390 3505 1
3391 3506 1 FUNCTION
3392 3507 1 This routine constructs a Primary Descriptor Sub-Node for a specified
3393 3508 1 symbol and appends this sub-node to a specified Primary Descriptor.
3394 3509 1 If the symbol is an array or a record, an Array or Record Sub-Node is
3395 3510 1 constructed and the specific information needed for those data types
3396 3511 1 is filled in. Otherwise, a Normal Sub-Node is constructed.
3397 3512 1
3398 3513 1 INPUTS
3399 3514 1 PRIMPTR - A pointer to the Root Node of the Primary Descriptor to
3400 3515 1 which the Sub-Node should be appended.
3401 3516 1
3402 3517 1 KIND - The KIND of the symbol for which the Primary Descriptor
3403 3518 1 Sub-Node should be constructed. This is the RST 'kind'
3404 3519 1 returned by DBG$STA_GETSYMBOL.
3405 3520 1
3406 3521 1 SYMID - The SYMID of the symbol for which the Primary Descriptor
3407 3522 1 Sub-Node should be constructed. If there is no SYMID (as
3408 3523 1 for an individual array element, for exmple), SYMID should
3409 3524 1 be zero.
3410 3525 1
3411 3526 1 FCODE - The FCODE ('format code') for the data type of the symbol for
3412 3527 1 which the Primary Descriptor Sub-Node should be constructed.
3413 3528 1 If the symbol is not a data items (i.e., if its KIND is not
3414 3529 1 RST$K_DATA or RST$K_TYPCOMP), FCODE should be zero.
3415 3530 1
3416 3531 1 TYPEID - The Type ID of the data item for which the Primary Descriptor
3417 3532 1 Sub-Node should be constructed. If the entity in question is
3418 3533 1 not a data item, TYPEID should be zero.
3419 3534 1
3420 3535 1 DSTPTR - A pointer to the DST record corresponding the the data item.
3421 3536 1 This is used in the case of BLISS data to obtain the
3422 3537 1 information about what kind of BLISS structure this is.
3423 3538 1
3424 3539 1 OUTPUTS
3425 3540 1 A Primary Descriptor Sub-Node is created and appended to the PRIMPTR
3426 3541 1 Primary Descriptor. PRIMPTR itself is not changed, however.
3427 3542 1 There is no other output.
3428 3543 1
3429 3544 1
3430 3545 2 BEGIN
3431 3546 2
3432 3547 2 MAP
3433 3548 2 DSTPTR: REF DST$RECORD, ! Pointer to DST record
3434 3549 2 PRIMPTR: REF DBG$PRIMARY, ! Pointer to Primary Descriptor
3435 3550 2 SYMID: REF RST$ENTRY; ! Pointer to symbol's RST entry
3436 3551 2
3437 3552 2 BUILTIN
3438 3553 2 INSQUE; ! Insert-Queue function
3439 3554 2
3440 3555 2 LITERAL
3441 3556 2 MAX_DIMS = 20; ! Maximum dimension count we allow
3442 3557 2
3443 3558 2 LOCAL
3444 3559 2 ATOMIC_TYPE, ! A dtype code
```



```
3445 3560 2 BIT_LENGTH,
3446 3561 2 BITSIZE,
3447 3562 2
3448 3563 2 BOUNDVEC: REF VECTOR[.LONG],
3449 3564 2
3450 3565 2 CELLTYPE,
3451 3566 2 COMP_VEC,
3452 3567 2 DIMVECPTR: REF VECTOR[.LONG],
3453 3568 2 DSCADDR: REF BLOCK[.BYTE],
3454 3569 2 HIGHPTR,
3455 3570 2 LENGTH,
3456 3571 2 LOWPTR,
3457 3572 2 NCOMPS,
3458 3573 2 NDIMS,
3459 3574 2 NODEPTR: REF DBG$PRIM_NODE,
3460 3575 2 OFFSET,
3461 3576 2 SIZE,
3462 3577 2 STRIDE: VECTOR[.MAX_DIMS, .LONG],
3463 3578 2
3464 3579 2 STRIDEPTR: REF VECTOR[.LONG],
3465 3580 2
3466 3581 2 STRIDE_SIZE,
3467 3582 2 SUB_TYPEID: REF RST$ENTRY,
3468 3583 2 SUBVECTOR:
3469 3584 2 REF DBG$PRIM_NODE_SUBS,
3470 3585 2 TMP_SYMID: REF RST$ENTRY;
3471 3586 2
3472 3587 2 LABEL
3473 3588 2 COMPUTE_STRIDES:
3474 3589 2
3475 3590 2
3476 3591 2 DBG$GL_CURRENT_PRIMARY = .PRIMPTR;
3477 3592 2
3478 3593 2 ! Determine what kind of symbol this is and allocate the Sub-Node memory
3479 3594 2 ! block accordingly. If this symbol is an array or a record, we also must
3480 3595 2 ! collect the special information needed for those data types and fill it
3481 3596 2 ! into the Primary Descriptor Sub-Node.
3482 3597 2
3483 3598 2 CASE .FCODE FROM RST$K_TYPE_MINIMUM TO RST$K_TYPE_MAXIMUM OF
3484 3599 2 SET
3485 3600 2
3486 3601 2
3487 3602 2 ! Handle arrays. Determine the number of dimensions in the array and
3488 3603 2 ! allocate an Array Sub-Node large enough to hold that many dimensions.
3489 3604 2 ! Then fill in all the fields specific to the Array Sub-Node.
3490 3605 2
3491 3606 2 [RST$K_TYPE_ARRAY]:
3492 3607 2 BEGIN
3493 3608 2
3494 3609 2
3495 3610 2 ! Get all needed type information about the array type from
3496 3611 2 ! DBG$STA_TYP_ARRAY. Allocate a memory block for the Array
3497 3612 2 ! Sub-Node and fill in the fields in the fixed part of that
3498 3613 2 ! Sub-Node.
3499 3614 2
3500 3615 2 DBG$STA_TYP_ARRAY(.TYPEID, DSCADDR, CELLTYPE,
3501 3616 2 NDIMS, DIMVECPTR, BITSIZE);
```

If we got a symid passed in to this routine, and the symid represents an array, try calling SYMVALUE to get an array descriptor for this array. If we get one, then use this descriptor instead of the one we got back from STA_TYP_ARRAY.

Note - normally, these 2 descriptors will be the same. However, for dynamic arrays in PASCAL, the runtime descriptor (which we get back when we call SYMVALUE with the symid) is correct, but the compile-time descriptor (which is part of the typespec) is wrong. This code is a workaround for this problem in the PASCAL DST.

```
IF .SYMID NEQ 0
THEN
  BEGIN
    LOCAL
      DESC: VECTOR[3],
      RSTPTR: REF RST$ENTRY,
      VALUE_KIND;
    RSTPTR = .SYMID;
    WHILE .RSTPTR[RST$B_KIND] NEQ RST$K_MODULE DO
      RSTPTR = .RSTPTR[RST$L_UPSCOPEPTR];
    IF .RSTPTR[RST$B_LANGUAGE] EQL DBG$K_PASCAL
    THEN
      BEGIN
        DBG$STA_SETCONTEXT(.SYMID);
        DBG$STA_SYMVALUE(.SYMID, DESC, VALUE_KIND);
        IF .VALUE_KIND EQL DBG$K_VAL_DESCR
        THEN
          DSCADDR = .DESC[0];
        END;
      END;
    IF .NDIMS GTR MAX_DIMS THEN SIGNAL(DBG$ TOOMANDIM);
    NODEPTR = DBG$GET_TEMPMEM(DBG$K_PRIM_SIZE_ARRAY +
      .NDIMS*DBG$K_PRIM_SIZE_SUBS);
    NODEPTR[DBG$B_PNARR_DIMCNT] = .NDIMS;

    *** The following is a temporary workaround to a problem in
    *** the PASCAL DST: They are giving us array descriptors with
    *** class UBA but dtype=0. Since dtype must be VU for this
    *** class, we fill in the correct dtype here in this case.
    IF .DSCADDR[DSC$B_CLASS] EQL DSC$K_CLASS_UBA
    THEN
      NODEPTR[DBG$B_PNARR_DTYPE] = DSC$K_DTYPE_VU
    ELSE
      *** End temporary workaround.
      NODEPTR[DBG$B_PNARR_DTYPE] = .DSCADDR[DSC$B_DTYPE];
    NODEPTR[DBG$B_PNARR_LENGTH] = .DSCADDR[DSC$B_LENGTH];
    NODEPTR[DBG$B_PNARR_SCALE] = .DSCADDR[DSC$B_SCALE];
```

```

NODEPTR[DBG$B_PNARR_DIGITS] = .DSCADDR[DSC$B_DIGITS];
NODEPTR[DBG$V_PNARR_COLUMN] = .DSCADDR[DSC$V_FL_COLUMN];
NODEPTR[DBG$L_PNARR_CELLTYPE] = .CELLTYPE;

! Set up pointers to the stride (or multiplier) vector and to the
! bounds vector in the array descriptor.
STRIDEPTR = .DSCADDR + 20;
BOUNDVEC = .STRIDEPTR + 4*.NDIMS;

! Determine what kind of array descriptor we have. Determine
! whether we have multipliers or strides and whether we are
! doing byte or bit addressing.
CASE .DSCADDR[DSC$B_CLASS] FROM DSC$K_CLASS_S TO DSC$K_CLASS_UBA OF
SET

! Handle an ordinary Array Descriptor. This descriptor has
! byte multipliers. Since we only allow strides in the Array
! Sub-Node, we convert the array multipliers into strides here.
! (Note that STRIDEPTR points to multipliers in this case.)
[DSC$K_CLASS_A]:
  COMPUTE_STRIDES: BEGIN

    *** Temporary workaround to a problem in the PL/I DST.
    *** The multipliers that we are getting in the array
    *** descriptors are not correct. They are giving us
    *** strides instead of multipliers. So we skip the
    *** computation of strides if language is PL/I.

    TMP_SYMID = .PRIMPTR[DBG$L_DHDR_SYMID0];
    IF .TMP_SYMID NEQ 0
    THEN
      BEGIN
        WHILE .TMP_SYMID[RST$B_KIND] NEQ RST$K_MODULE DO
          TMP_SYMID = .TMP_SYMID[RST$L_UPSCOPEPTR];
        IF (.TMP_SYMID[RST$B_LANGUAGE] EQL DBG$K_PLI) AND
          (.TMP_SYMID[RST$V_OLDPLIFLAG])
        THEN
          LEAVE COMPUTE_STRIDES;
        END;
      END;

    ! Pick up the array element length in bytes. We need this
    ! length to compute strides properly below.
    LENGTH = DBG$DATA_LENGTH(.DSCADDR);
    LENGTH = (.LENGTH + 7)/8;

    ! If this is a column-major order array, we compute the
    ! array's stride values from its multiplier values.
```

```

! IF .DSCADDR[DSC$V_FL_COLUMN]
! THEN
!   BEGIN
!     STRIDE[0] = .LENGTH;
!     INCR I FROM 1 TO .NDIMS - 1 DO
!       STRIDE[I] = .STRIDE[I - 1]*.STRIDEPTR[I - 1];
!     END
!
!   ! If this is a row-major order array, we compute the
!   ! strides in the opposite direction.
! ELSE
!   BEGIN
!     STRIDE[.NDIMS - 1] = .LENGTH;
!     DECR I FROM .NDIMS - 2 TO 0 DO
!       STRIDE[I] = .STRIDE[I + 1]*.STRIDEPTR[I + 1];
!     END;
!
!   ! Make STRIDEPTR point to the newly computed stride vector.
!   STRIDEPTR = STRIDE[0];
! END;
!
! Handle a Noncontiguous Array Descriptor or a Varying String
! Descriptor. Here we already have byte strides instead of
! multipliers so there is nothing we need to do.
[DSC$K_CLASS_NCA,
DSC$K_CLASS_VSA]:
0;
!
! Handle an Unaligned Bit Array Descriptor. This descriptor
! is like the Noncontiguous Array Descriptor except that it has
! bit strides instead of byte strides.
[DSC$K_CLASS_UBA]:
NODEPTR[DBG$V_PNARR_BITREF] = TRUE;
!
! Any other case constitutes an invalid array descriptor.
! Signal an error message.
[INRANGE, OVRANGE]:
SIGNAL(DBG$_INVARRDSC);
TES;
!
! Loop through the dimensions of the array to set up the subscript
! block-vector in the Array Sub-Node. In that vector, we set the
```



```
! subscript value to be the same as the lower bound--this is changed
! later when the actual subscript value is picked up. The lower and
! upper bounds, the stride, and the subscript type TYPEID are all
! picked up and stored away as well.
SUBVECTOR = NODEPTR(DBG$A_PNARR_SVECTOR);
OFFSET = 0;
INCR I FROM 0 TO .NDIMS - 1 DO
  BEGIN
    SUBVECTOR[I, DBG$L_PNSUB_SVALUE] = .BOUNDVEC[2*I];
    SUBVECTOR[I, DBG$L_PNSUB_STRIDE] = .STRIDEPTR[I];
    SUBVECTOR[I, DBG$L_PNSUB_LBOUND] = .BOUNDVEC[2*I];
    SUBVECTOR[I, DBG$L_PNSUB_UBOUND] = .BOUNDVEC[2*I + 1];

    SUB_TYPEID = 0;
    IF .DIMVECPTR[I] NEQ 0
    THEN
      BEGIN
        SUB_TYPEID = .DIMVECPTR[I];
        WHILE .SUB_TYPEID[RST$B_FCODE] EQL RST$K_TYPE_SUBRNG DO
          DBG$STA_TYP_SUBRNG(.SUB_TYPEID, SUB_TYPEID, LOWPTR, HIGHPTR, SIZE);

        SUBVECTOR[I, DBG$L_PNSUB_TYPEID] = .SUB_TYPEID;
      END
    ELSE
      SUBVECTOR[I, DBG$L_PNSUB_TYPEID] = .DIMVECPTR[I];

    OFFSET = .OFFSET - (.SUBVECTOR[I, DBG$L_PNSUB_STRIDE]*
      .BOUNDVEC[2*I]);

    ! One additional thing needs to be done for arrays indexed
    ! by enumeration types in ADA. The SVALUE field needs
    ! to be filled in to be the value of the enumeration,
    ! not its position.
    IF .DBG$GB_LANGUAGE EQL DBG$K_ADA
    THEN
      IF .SUB_TYPEID NEQ 0
      THEN
        IF .SUB_TYPEID[RST$B_FCODE] EQL RST$K_TYPE_ENUM
        THEN
          SUBVECTOR[I, DBG$L_PNSUB_SVALUE] =
            DBG$ENUM_VAL(.SUB_TYPEID,
              .SUBVECTOR[I, DBG$L_PNSUB_SVALUE]);
        END;
      END;

    ! Finally fill in the offset from the start of the array to
    ! element ARRAY[0,0,....,0]. Also mark symbol as an aggregate.
    NODEPTR(DBG$L_PNARR_OFFSET) = .OFFSET;
    IF .DBG$GL_ARRSUB_F[AG
    THEN
      PRIMPTR(DBG$V_DHDR_AGR) = TRUE;
```

```
3730
3731
3732
3733
3734
3735
3736
3737
3738
3739
3740
3741
3742
3743
3744
3745
3746
3747
3748
3749
3750
3751
3752
3753
3754
3755
3756
3757
3758
3759
3760
3761
3762
3763
3764
3765
3766
3767
3768
3769
3770
3771
3772
3773
3774
3775
3776
3777
3778
3779
3780
3781
3782
3783
3784
3785
3786
```

```
END;                                ! End of case for Array Sub-Node

! Handle records. Allocate a Record Sub-Node. Then simply initialize
! the record component index to be 1. If dot-qualification follows,
! this field will be changed to the actual component index of the
! selected record component. Also mark symbol as an aggregate.
[RST$K_TYPE_RECORD]:
BEGIN
  DBG$STA_TYP_RECORD(.TYPEID, NCOMPS, COMP_VEC, BITSIZE);
  NODEPTR = DBG$GET_TEMPMEM(DBG$K_PRIM_SIZE_RECORD);
  NODEPTR[DBG$W_PNREC_INDEX] = 1;
  NODEPTR[DBG$W_PNREC_NCOMPS] = .NCOMPS;
  IF .DBG$GL_REC_CMP_FLAG
  THEN
    PRIMPTR[DBG$V_DHDR_AGGR] = TRUE;
  END;

! Handle Record Variants (as in PASCAL or ADA). Here we allocate
! a Variant Sub-Node, but the fields specific to this sub-node are
! actually set by the caller (namely GET_RECORD_COMPONENT).
[RST$K_TYPE_VARIANT]:
  NODEPTR = DBG$GET_TEMPMEM(DBG$K_PRIM_SIZE_VARIANT);

! Handle BLISS data items. The four kinds of data that fall have this
! fcode are vectors, bitvectors, blocks, and blockvectors. For these
! four types, we have to go to the DST to obtain the information
! which is placed in the Primary Descriptor.
[RST$K_TYPE_BLI_DATA]:
BEGIN

  ! Assume aggregate until proved otherwise.
  PRIMPTR[DBG$V_DHDR_AGGR] = TRUE;

  ! Get the DST record if necessary.
  IF .DSTPTR EQL 0 AND .SYMID NEQ 0
  THEN
    DSTPTR = .SYMID [RST$L_DSTPTR];

  ! Check for the REF bit being set. If so, allocate a Primary
  ! Descriptor Normal Sub-Node and set the FCODE to pointer to
  ! indicate that dereferencing is taking place. Also turn off
  ! the aggregate flag for REF items.
  IF .DSTPTR [DST$V_BLI_REF] AND .SYMID NEQ 0
  THEN
```

```
BEGIN
NODEPTR = DBG$GET_TEMPMEM (DBG$K PRIM_SIZE_NORMAL);
PRIMPTR [DBG$V_DHDR_AGGR] = FALSE;
FCODE = RST$K_TYPE_PTR;
END
```

```
! If it is not a REF item, CASE on the kind of BLISS structure:
! Vector, Bitvector, Block, or Blockvector.
```

```
ELSE
```

```
BEGIN
CASE .DSTPTR [DST$V_BLI_STRUC] FROM DST$K_BLI_NOSTRUC
TO DST$K_BLI_BLKVEC OF
```

```
SET
```

```
! The "nostruc" code arises in the case where the structure of
! the data is a user-defined structure, and not one of the
! four built-ins. We treat such data as an ordinary variable.
```

```
[DST$K_BLI_NOSTRUC] :
```

```
BEGIN
PRIMPTR [DBG$V_DHDR_AGGR] = FALSE;
NODEPTR = DBG$GET_TEMPMEM (DBG$K PRIM_SIZE_NORMAL);
END;
```

```
! Handle vectors. We allocate enough space for an Array
! Sub-Node with one subscript. The fields of the subnode
! are then filled in.
```

```
[DST$K_BLI_VEC] :
```

```
BEGIN
FCODE = RST$K_TYPE_ARRAY;
NODEPTR = DBG$GET_TEMPMEM (DBG$K PRIM_SIZE_ARRAY +
DBG$K PRIM_SIZE_SUBS);
NODEPTR [DBG$B_PNARR_DIMCNT] = 1;
NODEPTR [DBG$B_PNARR_LENGTH] = .DSTPTR[DST$V_BLI_VEC_UNIT_SIZE];
```

```
!+
! Figure out the dtype based on the unit size and
! sign extension.
```

```
! VECTOR[BYTE] or VECTOR[BYTE,SIGNED]
```

```
IF .DSTPTR[DST$V_BLI_VEC_UNIT_SIZE] EQL 1
THEN
```

```
IF .DSTPTR[DST$V_BLI_VEC_SIGN_EXT] NEQ 0
THEN
```

```
NODEPTR [DBG$B_PNARR_DTYPE] = DSC$K_DTYPE_B
```

```
ELSE
```

```
NODEPTR [DBG$B_PNARR_DTYPE] = DSC$K_DTYPE_BU
```

```
! VECTOR[WORD] or VECTOR[WORD,SIGNED]
```

```
3787 3902 4
3788 3903 4
3789 3904 4
3790 3905 4
3791 3906 4
3792 3907 4
3793 3908 4
3794 3909 4
3795 3910 4
3796 3911 4
3797 3912 3
3798 3913 4
3799 3914 4
3800 3915 4
3801 3916 4
3802 3917 4
3803 3918 4
3804 3919 4
3805 3920 4
3806 3921 4
3807 3922 4
3808 3923 4
3809 3924 3
3810 3925 3
3811 3926 3
3812 3927 4
3813 3928 4
3814 3929 4
3815 3930 4
3816 3931 4
3817 3932 4
3818 3933 4
3819 3934 4
3820 3935 3
3821 3936 3
3822 3937 3
3823 3938 3
3824 3939 3
3825 3940 3
3826 3941 3
3827 3942 3
3828 3943 3
3829 3944 3
3830 3945 3
3831 3946 3
3832 3947 3
3833 3948 3
3834 3949 3
3835 3950 3
3836 3951 3
3837 3952 3
3838 3953 3
3839 3954 3
3840 3955 3
3841 3956 3
3842 3957 3
3843 3958 3
```

3844
3845
3846
3847
3848
3849
3850
3851
3852
3853
3854
3855
3856
3857
3858
3859
3860
3861
3862
3863
3864
3865
3866
3867
3868
3869
3870
3871
3872
3873
3874
3875
3876
3877
3878
3879
3880
3881
3882
3883
3884
3885
3886
3887
3888
3889
3890
3891
3892
3893
3894
3895
3896
3897
3898
3899
3900

3959
3960
3961
3962
3963
3964
3965
3966
3967
3968
3969
3970
3971
3972
3973
3974
3975
3976
3977
3978
3979
3980
3981
3982
3983
3984
3985
3986
3987
3988
3989
3990
3991
3992
3993
3994
3995
3996
3997
3998
3999
4000
4001
4002
4003
4004
4005
4006
4007
4008
4009
4010
4011
4012
4013
4014
4015

```
ELSE IF .DSTPTR[DST$V_BLI_VEC_UNIT_SIZE] EQL 2
THEN
  IF .DSTPTR[DST$V_BLI_VEC_SIGN_EXT] NEQ 0
  THEN
    NODEPTR [DBG$B_PNARR_DTYPE] = DSC$K_DTYPE_W
  ELSE
    NODEPTR [DBG$B_PNARR_DTYPE] = DSC$K_DTYPE_WU
  ! VECTOR[LONG] or VECTOR[LONG,SIGNED]
ELSE IF .DSTPTR[DST$V_BLI_VEC_UNIT_SIZE] EQL 4
THEN
  IF .DSTPTR[DST$V_BLI_VEC_SIGN_EXT] NEQ 0
  THEN
    NODEPTR [DBG$B_PNARR_DTYPE] = DSC$K_DTYPE_L
  ELSE
    NODEPTR [DBG$B_PNARR_DTYPE] = DSC$K_DTYPE_LU
  ! VECTOR[X] where X is not 1, 2, or 4
ELSE
  BEGIN
    IF .DSTPTR[DST$V_BLI_VEC_SIGN_EXT] NEQ 0
    THEN
      NODEPTR [DBG$B_PNARR_DTYPE] = DSC$K_DTYPE_V
    ELSE
      NODEPTR [DBG$B_PNARR_DTYPE] = DSC$K_DTYPE_VU;
    ! In this case the since the type has been set
    ! to bitfield then multiply the length by 8 to
    ! express it in bits.
    NODEPTR[DBG$W_PNARR_LENGTH] =
      .NODEPTR[DBG$W_PNARR_LENGTH] * 8;
    END;
  SUBVECTOR = NODEPTR [DBG$A_PNARR_SVECTOR];

  ! The stride can be either 1, 2, or 4, depending on whether
  ! the vector was declared as a byte, word, or longword
  ! vector. We obtain this information from the DST.
  SUBVECTOR [0, DBG$L_PNSUB_STRIDE] =
    .DSTPTR [DST$V_BLI_VEC_UNIT_SIZE];

  ! The upper bound on subscripts is one less than the
  ! number of units that were allocated in the declaration
  ! of the vector. (Origin-0 subscripting)
  IF .DSTPTR [DST$L_BLI_VEC_UNITS] EQL 0
  THEN
    SUBVECTOR [0, DBG$L_PNSUB_UBOUND] = 0
  ELSE
    SUBVECTOR [0, DBG$L_PNSUB_UBOUND] =
```



```

3901 4016
3902 4017
3903 4018
3904 4019
3905 4020
3906 4021
3907 4022
3908 4023
3909 4024
3910 4025
3911 4026
3912 4027
3913 4028
3914 4029
3915 4030
3916 4031
3917 4032
3918 4033
3919 4034
3920 4035
3921 4036
3922 4037
3923 4038
3924 4039
3925 4040
3926 4041
3927 4042
3928 4043
3929 4044
3930 4045
3931 4046
3932 4047
3933 4048
3934 4049
3935 4050
3936 4051
3937 4052
3938 4053
3939 4054
3940 4055
3941 4056
3942 4057
3943 4058
3944 4059
3945 4060
3946 4061
3947 4062
3948 4063
3949 4064
3950 4065
3951 4066
3952 4067
3953 4068
3954 4069
3955 4070
3956 4071
3957 4072

```

```

.DSTPTR [DST$B_BLI_VEC_UNITS]-1;

END;

! Handle bit-vectors. We allocate enough space for an
! Array Sub-Node with one subscript. The fields of the
! subnode are then filled in.
[DST$K_BLI_BITVEC] :
  BEGIN
    FCODE = RST$K_TYPE_ARRAY;
    NODEPTR = DBG$GET_TEMPMEM (DBG$K_PRIM_SIZE_ARRAY +
                                DBG$K_PRIM_SIZE_SUBS);

    ! Since this is a bitvector, set the BITREF flag.
    NODEPTR [DBG$V_PNARR_BITREF] = TRUE;
    NODEPTR [DBG$B_PNARR_DIMCNT] = 1;
    NODEPTR [DBG$B_PNARR_DTYPE] = DST$K_DTYPE_VU;
    NODEPTR [DBG$W_PNARR_LENGTH] = 1;
    SUBVECTOR = NODEPTR [DBG$A_PNARR_SVECTOR];
    SUBVECTOR [0, DBG$B_PNSUB_STRIDE] = 1;

    ! The upper bound on subscripts is one less than the
    ! number of units that were allocated in the declaration
    ! of the vector. (Origin-0 subscripting)
    IF .DSTPTR [DST$B_BLI_BITVEC_SIZE] EOL 0
    THEN
      SUBVECTOR [0, DBG$B_PNSUB_UBOUND] = 0
    ELSE
      SUBVECTOR [0, DBG$B_PNSUB_UBOUND] =
        .DSTPTR [DST$B_BLI_BITVEC_SIZE]-1;

    END;

! Handle blocks. We allocate enough space for an Array
! Sub-Node with one subscript. The fields of the subnode
! are then filled in.
[DST$K_BLI_BLOCK] :
  BEGIN
    PRIMPTR[DBG$V_DHDR_BLIBLK] = TRUE;
    FCODE = RST$K_TYPE_ARRAY;
    NODEPTR = DBG$GET_TEMPMEM (DBG$K_PRIM_SIZE_ARRAY +
                                DBG$K_PRIM_SIZE_SUBS);
    NODEPTR [DBG$B_PNARR_DIMCNT] = 1;
    SUBVECTOR = NODEPTR [DBG$A_PNARR_SVECTOR];

    ! Fill in stride and length as four. For purposes
    ! of aggregate examine, we are representing

```

```

3958 4073
3959 4074
3960 4075
3961 4076
3962 4077
3963 4078
3964 4079
3965 4080
3966 4081
3967 4082
3968 4083
3969 4084
3970 4085
3971 4086
3972 4087
3973 4088
3974 4089
3975 4090
3976 4091
3977 4092
3978 4093
3979 4094
3980 4095
3981 4096
3982 4097
3983 4098
3984 4099
3985 4100
3986 4101
3987 4102
3988 4103
3989 4104
3990 4105
3991 4106
3992 4107
3993 4108
3994 4109
3995 4110
3996 4111
3997 4112
3998 4113
3999 4114
4000 4115
4001 4116
4002 4117
4003 4118
4004 4119
4005 4120
4006 4121
4007 4122
4008 4123
4009 4124
4010 4125
4011 4126
4012 4127
4013 4128
4014 4129

```

```

: each element as a longword. If it turns out
: we are not doing aggregate examine, then
: we fix up this information in the GET_BLISS_SUBSCRIPTS
: routine.

```

```

SUBVECTOR [0, DBG$PNSUB_STRIDE] = 4;
NODEPTR [DBG$B_PNARR_DTYPE] = DSC$K_DTYPE_L;
NODEPTR [DBG$W_PNARR_LENGTH] = 4;

```

```

: The upper bound on subscripts is one less than the
: number of units that were allocated in the declaration
: of the vector. (Origin-0 subscripting)
: We temporarily dummy this up as if stride were 4, for
: purposes of aggregate output.

```

```

STRIDE_SIZE = .DSTPTR [DST$V_BLI_BLOCK_UNIT_SIZE];
IF (.DSTPTR [DST$L_BLI_BLOCK_UNITS] EQL 0) OR
   (.STRIDE_SIZE EQL 0)
THEN
    SUBVECTOR [0, DBG$PNSUB_UBOUND] = 0
ELSE
    SUBVECTOR [0, DBG$PNSUB_UBOUND] =
        (.DSTPTR [DST$L_BLI_BLOCK_UNITS]*.STRIDE_SIZE-1)
        /4;

```

```

END;

```

```

: Handle blockvectors. We allocate enough space for an
: Array Sub-Node with one subscript. The fields of the
: subnode are then filled in.

```

```

[DST$K_BLI_BLKVEC] :
BEGIN
    PRIMPTR[DBG$V_DHDR_BLIBLK] = TRUE;
    FCODE = RST$K_TYPE_ARRAY;
    NODEPTR = DBG$GET_TEMPMEM (DBG$K_PRIM_SIZE_ARRAY +
        2 * DBG$K_PRIM_SIZE_SUBS);
    NODEPTR [DBG$B_PNARR_DIMCNT] = 2;
    NODEPTR [DBG$B_PNARR_DTYPE] = DSC$K_DTYPE_V;
    NODEPTR [DBG$W_PNARR_LENGTH] = 32;
    SUBVECTOR = NODEPTR [DBG$A_PNARR_SVECTOR];

```

```

: The stride on the first subscript is the stride on
: the second subscript times the number of units per
: block. The upper bound on the first subscript
: depends on the number of blocks in the blockvector.

```

```

SUBVECTOR [0, DBG$PNSUB_STRIDE] =
    .DSTPTR [DST$B_BLI_BLKVEC_UNIT_SIZE] *
    .DSTPTR [DST$L_BLI_BLKVEC_UNITS];
IF .DSTPTR [DST$L_BLI_BLKVEC_BLOCKS] EQL 0
THEN
    SUBVECTOR [0, DBG$PNSUB_UBOUND] = 0

```

```

4015
4016
4017
4018
4019
4020
4021
4022
4023
4024
4025
4026
4027
4028
4029
4030
4031
4032
4033
4034
4035
4036
4037
4038
4039
4040
4041
4042
4043
4044
4045
4046
4047
4048
4049
4050
4051
4052
4053
4054
4055
4056
4057
4058
4059
4060
4061
4062
4063
4064
4065
4066
4067
4068
4069
4070
4071

```

```

ELSE
    SUBVECTOR [0, DBG$PNSUB_UBOUND] =
        .DSTPTR[.DST$C_BLI_BLKVEC_BLOCKS]-1;

    ! Fill in stride and length as four (we are always
    ! representing blocks as blocks of longwords).
    SUBVECTOR [1, DBG$PNSUB_STRIDE] = 4;
    NODEPTR [DBG$B_PNARR_DTYPE] = DSC$K_DTYPE_L;
    NODEPTR [DBG$W_PNARR_LENGTH] = 4;

    ! The upper bound on the subscript depends on the
    ! number of units per block.
    ! We temporarily dummy this up as if stride were 4, for
    ! purposes of aggregate output.
    STRIDE_SIZE = .DSTPTR [DST$B_BLI_BLKVEC_UNITS] SIZE;
    IF (.DSTPTR [DST$B_BLI_BLKVEC_UNITS] EQ 0) OR
        (.STRIDE_SIZE EQ 0)
    THEN
        SUBVECTOR [1, DBG$PNSUB_UBOUND] = 0
    ELSE
        SUBVECTOR [1, DBG$PNSUB_UBOUND] =
            (.DSTPTR [DST$B_BLI_BLKVEC_UNITS]*.STRIDE_SIZE-1)
            /4;

    END;

    TES;

    END;

    ! For bitvectors, vectors, blocks, and blockvectors, and REF stuff,
    ! make sure the celltype is filled in correctly.
    IF .FCODE EQ RST$K_TYPE_ARRAY
    THEN
        BEGIN
            ATOMIC_TYPE = .NODEPTR [DBG$B_PNARR_DTYPE];
            BIT_LENGTH = .NODEPTR [DBG$W_PNARR_LENGTH];
            IF (.ATOMIC_TYPE NEQ DSC$K_DTYPE_V) AND
                (.ATOMIC_TYPE NEQ DSC$K_DTYPE_VU) AND
                (.ATOMIC_TYPE NEQ DSC$K_DTYPE_SV) AND
                (.ATOMIC_TYPE NEQ DSC$K_DTYPE_SVU)
            THEN
                BIT_LENGTH = .BIT_LENGTH * 8;

            NODEPTR [DBG$PNSUB_CELLTYPE] =
                DBG$TYPEID_FOR_ATOMIC(.ATOMIC_TYPE, .BIT_LENGTH, FALSE);

        END;
    END;

```

```
4072 4187 3
4073 4188 3
4074 4189 4
4075 4190 4
4076 4191 4
4077 4192 4
4078 4193 3
4079 4194 3
4080 4195 3
4081 4196 3
4082 4197 3
4083 4198 3
4084 4199 3
4085 4200 3
4086 4201 3
4087 4202 3
4088 4203 3
4089 4204 3
4090 4205 3
4091 4206 3
4092 4207 3
4093 4208 3
4094 4209 3
4095 4210 3
4096 4211 3
4097 4212 3
4098 4213 3
4099 4214 3
4100 4215 3
4101 4216 3
4102 4217 3
4103 4218 3
4104 4219 3
4105 4220 3
4106 4221 3
4107 4222 3
4108 4223 3
4109 4224 3
4110 4225 3
4111 4226 3
4112 4227 3
4113 4228 3
4114 4229 3
4115 4230 3
4116 4231 3
4117 4232 3
4118 4233 3
4119 4234 1

IF .FCODE EQL RST$K_TYPE_PTR
THEN
  BEGIN
    ATOMIC_TYPE = DSC$K_DTYPE_L;
    BIT_LENGTH = 32;
    TYPEID = DBG$TYPEID_FOR_ATOMIC(.ATOMIC_TYPE, .BIT_LENGTH, FALSE);
  END;
END;

! For all other cases, just allocate a Normal Sub-Node. Also
! clear the aggregate flag.
[INRANGE, OVRANGE]:
  BEGIN
    NODEPTR = DBG$GET_TEMPMEM(DBG$K_PRIM_SIZE_NORMAL);
    PRIMPTR[DBG$V_DHDR_AGGR] = FALSE;
  END;
TES;

! Fill in the standard fields common to all Primary Descriptor Sub-Nodes.
NODEPTR[DBG$B_PNODE_KIND] = .KIND;
NODEPTR[DBG$B_PNODE_FCODE] = .FCODE;
NODEPTR[DBG$B_PNODE_TYPEID] = .TYPEID;
NODEPTR[DBG$B_PNODE_SYMID] = .SYMID;
NODEPTR[DBG$B_PNODE_RELOC] = 0;

! Also set the final Sub-Node's KIND, FCODE, and TYPEID in the Primary
! Descriptor Root Node. The Root Node thus describes the object described
! by the Primary Symbol as a whole.
PRIMPTR[DBG$B_DHDR_KIND] = .KIND;
PRIMPTR[DBG$B_DHDR_FCODE] = .FCODE;
PRIMPTR[DBG$B_DHDR_TYPEID] = .TYPEID;

! Append the Sub-Node to the Primary Descriptor by linking it in at the
! end of the doubly linked Sub-Node chain. Then return.
INSQUE(.NODEPTR, .PRIMPTR[DBG$B_PRIM_BLINK]);
RETURN;
END;
```

```
OFFC 00000
SE FF74 CE 9E 00002
58 04 AC D0 00007
```

```
.ENTRY DBG$BUILD_PRIMARY_SUBNODE, Save R2,R3,R4,- : 3503
R5,R6,R7,R8,R9,R10,R11
MOVAB -140(SP), SP
MOVL PRIMPTR, R8 : 3591
```


002C	15	00000000G	00	10	58	DO	0000B	MOVL	R8, DBG\$GL_CURRENT_PRIMARY		
002C	002C		01		AC	CF	00012	CASEL	FCODE, #1, #21		3598
002C	025B		002C		003F		00017	.WORD	3\$-1\$,-		
002C	002C		002C		002C		0001F		2\$-1\$,-		
002C	002C		002C		002C		00027		2\$-1\$,-		
002C	002C		002C		029F		0002F		2\$-1\$,-		
002C	0290		002C		002C		00037		2\$-1\$,-		
			002C		002C		0003F		2\$-1\$,-		
									32\$-1\$,-		
									2\$-1\$,-		
									2\$-1\$,-		
									2\$-1\$,-		
									2\$-1\$,-		
									2\$-1\$,-		
									2\$-1\$,-		
									36\$-1\$,-		
									2\$-1\$,-		
									2\$-1\$,-		
									2\$-1\$,-		
									2\$-1\$,-		
									2\$-1\$,-		
									2\$-1\$,-		
									34\$-1\$,-		
									2\$-1\$,-		
									2\$-1\$,-		
									2\$-1\$		
					06	DD	00043	2\$:	PUSHL	#6	4203
		00000000G	00		01	FB	00045		CALLS	#1, DBG\$GET_TEMPMEM	
			55		50	DO	0004C		MOVL	R0, NODEPTR	
		04	A8		01	8A	0004F		BICB2	#1, 4(R8)	4204
					047B	31	00053		BRW	66\$	3598
				24	AE	9F	00056	3\$:	PUSHAB	BITSIZE	3615
				04	AE	9F	00059		PUSHAB	DIMVECPTR	
				0C	AE	9F	0005C		PUSHAB	NDIMS	
				14	AE	9F	0005F		PUSHAB	CELLTYPE	
				1C	AE	9F	00062		PUSHAB	DSCADDR	
				14	AC	DD	00065		PUSHL	TYPEID	
		00000000G	00		06	FB	00068		CALLS	#6, DBG\$STA_TYP_ARRAY	
			52	0C	AC	DO	0006F		MOVL	SYMID, R2	3631
					38	13	00073		BEQL	6\$	
			50		52	DO	00075		MOVL	R2, RSTPTR	3638
			01	14	A0	91	00078	4\$:	CMPB	20(RSTPTR), #1	3639
					06	13	0007C		BEQL	5\$	
			50	10	A0	DO	0007E		MOVL	16(RSTPTR), RSTPTR	3640
					F4	11	00082		BRB	4\$	
			06	29	A0	91	00084	5\$:	CMPB	41(RSTPTR), #6	3641
					23	12	00088		BNEQ	6\$	
					52	DD	0008A		PUSHL	R2	3644
		00000000G	00		01	FB	0008C		CALLS	#1, DBG\$STA_SETCONTEXT	
				10	AE	9F	00093		PUSHAB	VALUE_KIND	3645
				34	AE	9F	00096		PUSHAB	DESC	
					52	DD	00099		PUSHL	R2	
		00000000G	00		03	FB	0009B		CALLS	#3, DBG\$STA_SYMVALUE	
			03	10	AE	D1	000A2		CMPL	VALUE_KIND, #3	3646
					05	12	000A6		BNEQ	6\$	
				30	AE	DO	000A8		MOVL	DESC, DSCADDR	3648
		0C	AE	04	AE	DO	000AD	6\$:	MOVL	NDIMS, R9	3652
			59		59	D1	000B1		CMPL	R9, #20	
			14		0D	15	000B4		BLEQ	7\$	

[illegible]

3C AE40 F2	38 AE40 50	FC A340 51	C5 00181 16\$:	MULL3	-4(STRIDEPTR)[I], STRIDE-4[I], STRIDE[I]	3737
		18	F3 0018B 17\$:	AOBLEQ	R1, I, 16\$	
		50	11 0018F	BRB	21\$	3732
3C AE41 50		50	D0 00191 18\$:	MOVL	LENGTH, STRIDE[R1]	3747
		FF A9	9E 00196	MOVAB	-1(R9), I	3748
		0A 11	0019A	BRB	20\$	
3C AE40	40 AE40 F3	04 A340	C5 0019C 19\$:	MULL3	4(STRIDEPTR)[I], STRIDE+4[I], STRIDE[I]	3749
	53	50	F4 001A6 20\$:	SOBGEQ	I, 19\$	
		3C AE	9E 001A9 21\$:	MOVAB	STRIDE, STRIDEPTR	3756
		04 11	001AD	BRB	23\$	3690
	0A A5	04 88	001AF 22\$:	BISB2	#4, 10(NODEPTR)	3774
	54	28 A5	9E 001B3 23\$:	MOVAB	40(R5), SUBVECTOR	3793
		5B D4	001B7	CLRL	OFFSET	3794
	52	01 CE	001B9	MNEGL	#1, I	3803
		009D 31	001BC	BRW	29\$	
57	52	14 C5	001BF 24\$:	MULL3	#20, I, R7	3797
56	52	01 78	001C3	ASHL	#1, I, R6	
		6744 9F	001C7	PUSHAB	(R7)[SUBVECTOR]	
	9E	6A46 D0	001CA	MOVL	(BOUNDVEC)[R6], a(SP)+	
		04 A744 9F	001CE	PUSHAB	4(R7)[SUBVECTOR]	3798
	9E	6342 D0	001D2	MOVL	(STRIDEPTR)[I], a(SP)+	
		08 A744 9F	001D6	PUSHAB	8(R7)[SUBVECTOR]	3799
	9E	6A46 D0	001DA	MOVL	(BOUNDVEC)[R6], a(SP)+	
		0C A744 9F	001DE	PUSHAB	12(R7)[SUBVECTOR]	3800
	9E	04 AA46 D0	001E2	MOVL	4(BOUNDVEC)[R6], a(SP)+	
		20 AE D4	001E7	CLRL	SUB_TYPEID	3802
	50	00 BE42 D0	001EA	MOVL	aDI[VECPTR][I], R0	3803
		2F 13	001EF	BEQL	27\$	
	20 AE	50 D0	001F1	MOVL	R0, SUB_TYPEID	3806
	50	20 AE D0	001F5 25\$:	MOVL	SUB_TYPEID, R0	3807
	09	18 A0	91 001F9	CMPB	24(R0), #9	
		17 12	001FD	BNEQ	26\$	
		14 AE 9F	001FF	PUSHAB	SIZE	3808
		1C AE 9F	00202	PUSHAB	HIGHTPTR	
		24 AE 9F	00205	PUSHAB	LOWPTR	
		2C AE 9F	00208	PUSHAB	SUB_TYPEID	
		50 DD	0020B	PUSHL	R0	
00000000G 00		05 FB	0020D	CALLS	#5, DBG\$STA_TYP_SUBRNG	
		DF 11	00214	BRB	25\$	
	9E	10 A744 9F	00216 26\$:	PUSHAB	16(R7)[SUBVECTOR]	3810
		24 AE D0	0021A	MOVL	SUB_TYPEID, a(SP)+	
		07 11	0021E	BRB	28\$	3803
		10 A744 9F	00220 27\$:	PUSHAB	16(R7)[SUBVECTOR]	3814
	9E	50 D0	00224	MOVL	R0, a(SP)+	
		04 A744 9F	00227 28\$:	PUSHAB	4(R7)[SUBVECTOR]	3817
56	9E	6A46 C5	0022B	MULL3	(BOUNDVEC)[R6], a(SP)+, R6	
	5B	56 C2	00230	SUBL2	R6, OFFSET	3816
	09 00000000G 00	91	00233	CMPB	DBG\$GB_LANGUAGE, #9	3825
		20 12	0023A	BNEQ	29\$	
	50	20 AE D0	0023C	MOVL	SUB_TYPEID, R0	3827
		1A 13	00240	BEQL	29\$	
	04	18 A0	91 00242	CMPB	24(R0), #4	3829
		14 12	00246	BNEQ	29\$	
		6744 9F	00248	PUSHAB	(R7)[SUBVECTOR]	3833
		9E DD	0024B	PUSHL	a(SP)+	
		50 DD	0024D	PUSHL	R0	3832
00000000G 00		02 FB	0024F	CALLS	#2, DBG\$ENUM_VAL	

PC	Op	Op2	Op3	Op4	Op5	Op6	Op7	Op8	Op9	Op10	Op11	Op12	Op13	Op14	Op15	Op16	Op17	Op18	Op19	Op20	Op21	Op22	Op23	Op24	Op25	Op26	Op27	Op28	Op29	Op30	Op31	Op32	Op33	Op34	Op35	Op36	Op37	Op38	Op39	Op40	Op41	Op42	Op43	Op44	Op45	Op46	Op47	Op48	Op49	Op50	Op51	Op52	Op53	Op54	Op55	Op56	Op57	Op58	Op59	Op60	Op61	Op62	Op63	Op64	Op65	Op66	Op67	Op68	Op69	Op70	Op71	Op72	Op73	Op74	Op75	Op76	Op77	Op78	Op79	Op80	Op81	Op82	Op83	Op84	Op85	Op86	Op87	Op88	Op89	Op90	Op91	Op92	Op93	Op94	Op95	Op96	Op97	Op98	Op99	Op100	Op101	Op102	Op103	Op104	Op105	Op106	Op107	Op108	Op109	Op110	Op111	Op112	Op113	Op114	Op115	Op116	Op117	Op118	Op119	Op120	Op121	Op122	Op123	Op124	Op125	Op126	Op127	Op128	Op129	Op130	Op131	Op132	Op133	Op134	Op135	Op136	Op137	Op138	Op139	Op140	Op141	Op142	Op143	Op144	Op145	Op146	Op147	Op148	Op149	Op150	Op151	Op152	Op153	Op154	Op155	Op156	Op157	Op158	Op159	Op160	Op161	Op162	Op163	Op164	Op165	Op166	Op167	Op168	Op169	Op170	Op171	Op172	Op173	Op174	Op175	Op176	Op177	Op178	Op179	Op180	Op181	Op182	Op183	Op184	Op185	Op186	Op187	Op188	Op189	Op190	Op191	Op192	Op193	Op194	Op195	Op196	Op197	Op198	Op199	Op200	Op201	Op202	Op203	Op204	Op205	Op206	Op207	Op208	Op209	Op210	Op211	Op212	Op213	Op214	Op215	Op216	Op217	Op218	Op219	Op220	Op221	Op222	Op223	Op224	Op225	Op226	Op227	Op228	Op229	Op230	Op231	Op232	Op233	Op234	Op235	Op236	Op237	Op238	Op239	Op240	Op241	Op242	Op243	Op244	Op245	Op246	Op247	Op248	Op249	Op250	Op251	Op252	Op253	Op254	Op255	Op256	Op257	Op258	Op259	Op260	Op261	Op262	Op263	Op264	Op265	Op266	Op267	Op268	Op269	Op270	Op271	Op272	Op273	Op274	Op275	Op276	Op277	Op278	Op279	Op280	Op281	Op282	Op283	Op284	Op285	Op286	Op287	Op288	Op289	Op290	Op291	Op292	Op293	Op294	Op295	Op296	Op297	Op298	Op299	Op300	Op301	Op302	Op303	Op304	Op305	Op306	Op307	Op308	Op309	Op310	Op311	Op312	Op313	Op314	Op315	Op316	Op317	Op318	Op319	Op320	Op321	Op322	Op323	Op324	Op325	Op326	Op327	Op328	Op329	Op330	Op331	Op332	Op333	Op334	Op335	Op336	Op337	Op338	Op339	Op340	Op341	Op342	Op343	Op344	Op345	Op346	Op347	Op348	Op349	Op350	Op351	Op352	Op353	Op354	Op355	Op356	Op357	Op358	Op359	Op360	Op361	Op362	Op363	Op364	Op365	Op366	Op367	Op368	Op369	Op370	Op371	Op372	Op373	Op374	Op375	Op376	Op377	Op378	Op379	Op380	Op381	Op382	Op383	Op384	Op385	Op386	Op387	Op388	Op389	Op390	Op391	Op392	Op393	Op394	Op395	Op396	Op397	Op398	Op399	Op400	Op401	Op402	Op403	Op404	Op405	Op406	Op407	Op408	Op409	Op410	Op411	Op412	Op413	Op414	Op415	Op416	Op417	Op418	Op419
----	----	-----	-----	-----	-----	-----	-----	-----	-----	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------

		00000000G	00	0F	DD	0031D	PUSHL	#15	3937
			55	01	FB	0031F	CALLS	#1, DBG\$GET_TEMPHEM	
			51	50	DD	00326	MOVL	R0, NODEPTR	
		03	A1	A5	9E	00329	MOVAB	24(NODEPTR), R1	3939
			50	01	90	0032D	MOVB	#1, 3(R1)	
S3	60		04	A2	9E	00331	MOVAB	10(R2), R0	3940
		1C	A5	00	EF	00335	EXTZV	#0, #4, (R0), R3	
01	60		04	53	B0	0033A	MOVW	R3, 28(NODEPTR)	
				00	ED	0033E	CMPZV	#0, #4, (R0), #1	3949
		F0	8F	12	12	00343	BNEQ	44\$	
				60	93	00345	BITB	(R0), #240	3951
		02	A1	06	13	00349	BEQL	43\$	
				06	90	0034B	MOVB	#6, 2(R1)	3953
		02	A1	4C	11	0034F	BRB	51\$	
				02	90	00351	MOVW	#2, 2(R1)	3955
				46	11	00355	BRB	51\$	3951
02	60		04	00	ED	00357	CMPZV	#0, #4, (R0), #2	3959
				12	12	0035C	BNEQ	46\$	
		F0	8F	60	93	0035E	BITB	(R0), #240	3961
				06	13	00362	BEQL	45\$	
		02	A1	07	90	00364	MOVB	#7, 2(R1)	3963
				33	11	00368	BRB	51\$	
		02	A1	03	90	0036A	MOVW	#3, 2(R1)	3965
				2D	11	0036E	BRB	51\$	3961
04	60		04	00	ED	00370	CMPZV	#0, #4, (R0), #4	3969
				12	12	00375	BNEQ	48\$	
		F0	8F	60	93	00377	BITB	(R0), #240	3971
				06	13	0037B	BEQL	47\$	
		02	A1	08	90	0037D	MOVB	#8, 2(R1)	3973
				1A	11	00381	BRB	51\$	
		02	A1	04	90	00383	MOVW	#4, 2(R1)	3975
				14	11	00387	BRB	51\$	3971
		F0	8F	60	93	00389	BITB	(R0), #240	3981
				06	13	0038D	BEQL	49\$	
		02	A1	01	90	0038F	MOVB	#1, 2(R1)	3983
				04	11	00393	BRB	50\$	
		02	A1	22	90	00395	MOVW	#34, 2(R1)	3985
		1C	A5	08	A4	00399	MULW2	#8, 28(NODEPTR)	3992
			54	A5	9E	0039D	MOVAB	40(R5), SUBVECTOR	3995
04	A4	60	04	00	EF	003A1	EXTZV	#0, #4, (R0), 4(SUBVECTOR)	4003
				24	11	003A7	BRB	53\$	4010
		10	AC	01	DD	003A9	MOVL	#1, FCODE	4027
				0F	DD	003AD	PUSHL	#15	4028
		00000000G	00	01	FB	003AF	CALLS	#1, DBG\$GET_TEMPHEM	
			55	50	DD	003B6	MOVL	R0, NODEPTR	
		0A	A5	04	88	003B9	BISB2	#4, 10(NODEPTR)	4034
		1A	A5	8F	DD	003BD	MOVL	#65826, 26(NODEPTR)	4036
			54	A5	9E	003C5	MOVAB	40(R5), SUBVECTOR	4038
		04	A4	01	DD	003C9	MOVL	#1, 4(SUBVECTOR)	4039
				A2	D5	003CD	TSTL	6(R2)	4046
				3A	13	003D0	BEQL	55\$	
		0C	A4	01	C3	003D2	SUBL3	#1, 6(R2), 12(SUBVECTOR)	4052
				43	11	003D8	BRB	57\$	3914
			63	10	88	003DA	BISB2	#16, (R3)	4063
		10	AC	01	DD	003DD	MOVL	#1, FCODE	4064
				0F	DD	003E1	PUSHL	#15	4065
		00000000G	00	01	FB	003E3	CALLS	#1, DBG\$GET_TEMPHEM	

50	0A	A2	04 1A	55 54 A4 A5 04	00040108	28 06 0C	50 A5 04 8F 00 A2 04 50 05 A4 71 50 51 04 63 10 01 14 01 50 8F A5 0E A2 0A 06 0C 06 01 04 08 04 05 A4 06 01 04 08 04 05 A4 0C 0C 06 1A 1C 0E 0A 20 0C 01 2E A5 52 12 52 0D 52 08 52 03 08 7E 0C	D0 9E D0 D0 EF D5 13 D5 12 D4 11 C5 D7 C7 11 88 D0 DD FB D0 D0 04 9E 9A C5 D5 12 D4 11 C3 D0 90 B0 9A D5 13 D5 12 D4 11 C5 D7 C7 D1 12 A5 3C D1 13 D1 13 D1 13 D1 13 C4 D4 BB	003EA 003ED 003F1 003F5 003FD 00403 00406 00408 0040A 0040C 0040F 00411 00416 00418 0041D 00422 00426 00428 0042F 00432 0043A 0043E 00442 00448 0044B 0044D 00450 00452 00458 0045C 00460 00464 00468 0046B 0046D 0046F 00471 00474 00476 00478 0047D 00482 00486 00488 0048C 00490 00493 00495 00498 0049A 0049D 0049F 004A2 004A4 004A7 004A9	MOVL MOVAB MOVL MOVL EXTZV TSTL BEQL TSTL BNEQ CLRL BRB MULL3 DECL DIVL3 BRB BISB2 MOVL PUSHL CALLS MOVL MOVL MOVAB MOVZBL MULL3 TSTL BNEQ CLRL BRB SUBL3 MOVL MOVB MOVW MOVZBL TSTL BEQL TSTL BNEQ CLRL BRB MULL3 DECL DIVL3 CMPL BNEQ MOVZBL MOVZWL CMPL BEQL CMPL BEQL CMPL BEQL MULL2 CLRL PUSHR	R0, NODEPTR 40(R5), SUBVECTOR #4, 4(SUBVECTOR) #262408, 26(NODEPTR) #0, #4, 10(R2), STRIDE_SIZE 6(R2) 55\$ STRIDE_SIZE 56\$ 12(SUBVECTOR) 63\$ STRIDE_SIZE, 6(R2), R1 R1 #4, R1, 12(SUBVECTOR) 63\$ #16, (R3) #1, FCODE #20 #1, DBG\$GET_TEMPMEM R0, NODEPTR #2097665, 26(NODEPTR) 40(R5), SUBVECTOR 14(R2), R1 10(R2), R1, 4(SUBVECTOR) 6(R2) 59\$ 12(SUBVECTOR) 60\$ #1, 6(R2), 12(SUBVECTOR) #4, 24(SUBVECTOR) #8, 26(NODEPTR) #4, 28(NODEPTR) 14(R2), STRIDE_SIZE 10(R2) 61\$ STRIDE_SIZE 62\$ 32(SUBVECTOR) 63\$ STRIDE_SIZE, 10(R2), R2 R2 #4, R2, 32(SUBVECTOR) FCODE, #1 65\$ 26(NODEPTR), ATOMIC_TYPE 28(NODEPTR), BIT_LENGTH ATOMIC_TYPE, #1 64\$ ATOMIC_TYPE, #34 64\$ ATOMIC_TYPE, #41 64\$ ATOMIC_TYPE, #42 64\$ #8, BIT_LENGTH -(SP) #*M(R2,R3)	4068 4078 4079 4089 4090 4091 4093 4097 4098 3914 4109 4110 4111 4114 4116 4126 4127 4129 4133 4139 4140 4141 4149 4150 4151 4153 4157 4158 4170 4173 4174 4175 4176 4177 4178 4180 4183
----	----	----	----------	----------------------------	----------	----------------	--	--	--	--	---	--

00000000G	00		03	FB	004AB		CALLS	#3, DBG\$TYPEID_FOR_ATOMIC	
24	A5		50	D0	004B2		MOVL	R0, 36(NODEPTR)	
	10	10	AC	D1	004B6	65\$:	CMPL	FCODE, #16	4187
	52		15	12	004BA		BNEQ	66\$	
	53		08	D0	004BC		MOVL	#8, ATOMIC_TYPE	4190
			20	D0	004BF		MOVL	#32, BIT_LENGTH	4191
			7E	D4	004C2		CLRL	-(SP)	4192
			0C	BB	004C4		PUSHR	#*M<R2,R3>	
00000000G	00		03	FB	004C6		CALLS	#3, DBG\$TYPEID_FOR_ATOMIC	
14	AC		50	D0	004CD		MOVL	R0, TYPEID	
08	A5	08	AC	90	004D1	66\$:	MOVB	KIND, 8(NODEPTR)	4212
09	A5	10	AC	90	004D6		MOVB	FCODE, 9(NODEPTR)	4213
0C	A5	14	AC	D0	004DB		MOVL	TYPEID, 12(NODEPTR)	4214
10	A5	0C	AC	D0	004E0		MOVL	SYMID, 16(NODEPTR)	4215
		14	A5	D4	004E5		CLRL	20(NODEPTR)	4216
07	A8	08	AC	90	004E8		MOVB	KIND, 7(R8)	4223
06	A8	10	AC	90	004ED		MOVB	FCODE, 6(R8)	4224
08	A8	14	AC	D0	004F2		MOVL	TYPEID, 8(R8)	4225
18	BB		65	0E	004F7		INSQUE	(NODEPTR), 324(R8)	4231
			04	004FB			RET		4234

; Routine Size: 1276 bytes, Routine Base: DBG\$CODE + 006C

```
4121 4235 1 GLOBAL ROUTINE DBG$EXP_INT(INPUT_DESC, RADIX, VALUE_PTR, TERM_INDEX) =
4122 4236 1
4123 4237 1 FUNCTION
4124 4238 1 This is the common Expression Interpreter for most languages
4125 4239 1 supported by DEBUG. It parses and evaluates a source language
4126 4240 1 expression and returns a Value Descriptor which represents the
4127 4241 1 value of the expression. This routine itself is only a set-up
4128 4242 1 routine which sets up the character pointer and the expression
4129 4243 1 radix to use and then calls DBG$EXPRESSION_PARSER to do the
4130 4244 1 actual work.
4131 4245 1
4132 4246 1 INPUTS
4133 4247 1 INPUT_DESC - The address of a VAX standard string descriptor which
4134 4248 1 describes the input string to be parsed. The length is
4135 4249 1 actually not used, however--the string is instead assumed
4136 4250 1 to be terminated by a carriage-return character.
4137 4251 1
4138 4252 1 RADIX - The radix to be used to interpret integer constants in the
4139 4253 1 input string. The allowed radix values are DBG$K_DECIMAL,
4140 4254 1 DBG$K_HEX, DBG$K_OCTAL, and DBG$K_BINARY.
4141 4255 1
4142 4256 1 VALUE_PTR - The address of a longword to receive a pointer to the
4143 4257 1 value descriptor returned by this routine as its output.
4144 4258 1
4145 4259 1 TERM_INDEX - A "terminator index" which indicates which lexical tokens
4146 4260 1 are allowed as expression terminators in this context. These
4147 4261 1 index values have names of the form TOKEN$K_TERM_xxx.
4148 4262 1
4149 4263 1 5th Optional Parameter - If this is present, and the value is
4150 4264 1 DBG$K_DEPOSIT_VERB then pass this into
4151 4265 1 DBG$EXPRESSION_PARSER, so that in
4152 4266 1 DBG$EXPRESSION_PARSER, when the expression is not
4153 4267 1 address expression and in deposit command,
4154 4268 1 DBG$EVAL LANG OPERATOR will not be called with
4155 4269 1 DBG$GL_IDENTITY_TOKEN.
4156 4270 1 (This is passed in from DBG$NPARSE_DEPOSIT, from
4157 4271 1 DBG$NPARSE_EXPRESSION).
4158 4272 1
4159 4273 1 OUTPUTS
4160 4274 1 VALUE_PTR - The address of a Value Descriptor is returned to VALUE_PTR.
4161 4275 1 This Value Descriptor represents the value of the expression
4162 4276 1 interpreted by DBG$EXP_INT.
4163 4277 1
4164 4278 1 The value ST$K_SUCCESS is return as the routine result if the expres-
4165 4279 1 sion was terminated by a carriage-return character. If it
4166 4280 1 was terminated any other way (i.e., by a terminator token),
4167 4281 1 the value ST$K_WARNING is returned.
4168 4282 1
4169 4283 1
4170 4284 2 BEGIN
4171 4285 2
4172 4286 2 MAP
4173 4287 2 INPUT_DESC: REF BLOCK[ BYTE], ! Pointer to input string descriptor
4174 4288 2 VALUE_PTR: REF VECTOR[1]; ! Longword to receive Value Descr ptr.
4175 4289 2
4176 4290 2 LOCAL
4177 4291 2 VALPTR: REF DBG$VALDESC; ! Pointer to returned Value Descriptor
```



```
4178
4179
4180
4181
4182
4183
4184
4185
4186
4187
4188
4189
4190
4191
4192
4193
4194
4195
4196
4197
4198
4199
4200
4201
4202
4203
4204
4205
4206
4207
4208
4209
4210
4211
4212
4213
4214
4215
4216
4217
4218
4219
4220
4221
```

```
4292
4293
4294
4295
4296
4297
4298
4299
4300
4301
4302
4303
4304
4305
4306
4307
4308
4309
4310
4311
4312
4313
4314
4315
4316
4317
4318
4319
4320
4321
4322
4323
4324
4325
4326
4327
4328
4329
4330
4331
4332
4333
4334
4335
```

BUILTIN ACTUALCOUNT,ACTUALPARAMETER;

! Set up CHARPTR to point to the start of the expression string. Also set
! up the radix we are to use in the scan and initialize some variables.

CHARPTR = .INPUT_DESC[DSC\$A_POINTER];
EXPRESSION_RADIX = .RADIX;
SAVED_TOKEN = 0;

! Call the Expression Parser to parse the language expression. If the
! result is an unconverted constant Value Descriptor, we convert it to
! a real Value Descriptor here. Then return the Value Descriptor pointer.

IF ACTUALCOUNT() GTR 4
THEN

BEGIN
IF ACTUALPARAMETER(5) NEQ DBG\$K_DEPOSIT_VERB
THEN
\$DBG_ERROR('DBGPARSER\DBG\$EXP_INT');

VALPTR = DBG\$EXPRESSION_PARSER(FALSE,
.TERM_POINTER_TBL[.TERM_INDEX] + TABLEBASE,
DBG\$K_DEPOSIT_VERB);

END

ELSE
VALPTR = DBG\$EXPRESSION_PARSER(FALSE,
.TERM_POINTER_TBL[.TERM_INDEX] + TABLEBASE);
VALUE_PTR[0] = .VALPTR;

! Fix up the string descriptor to reflect the new location of the parse
! pointer and return the appropriate status code.

INPUT_DESC[DSC\$W_LENGTH] = .INPUT_DESC[DSC\$W_LENGTH] +
.INPUT_DESC[DSC\$A_POINTER] - .CHARPTR;
INPUT_DESC[DSC\$A_POINTER] = .CHARPTR;
IF .CHARPTR[0] NEQ CAR_RET THEN RETURN ST\$K_WARNING;
RETURN ST\$K_SUCCESS;

END;

```
24 47 42 44 5C 52 45 53 52 41 50 47 42 44 15 03040 P.AWR: .PSECT DBG$PLIT, NOWRT, SHR, PIC, 0
54 4E 49 5F 50 58 45 0304F .ASCII <21>\DBGPARSER\<92>\DBG$EXP_INT\
```

003C 00000

.PSECT DBG\$CODE, NOWRT, SHR, PIC, 0
.ENTRY DBG\$EXP_INT, Save R2, R3, R4, R5

: 4235

	55	00000000	EF	9E	00002	MOVAB	CHARPTR, R5	...
	54	00000000	EF	9E	00009	MOVAB	TERM_POINTER_TBL, R4	...
	53	04	AC	D0	00010	MOVL	INPUT_DESC, R3	4299
	65	04	A3	D0	00014	MOVL	4(R3), CHARPTR	...
040C	C5	08	AC	D0	00018	MOVL	RADIX, EXPRESSION_RADIX	4300
		0424	C5	D4	0001E	CLRL	SAVED_TOKEN	4301
	52	10	AC	D0	00022	MOVL	TERM_INDEX, R2	4316
	04		6C	91	00026	CMPB	(AP), #4	4308
			30	1B	00029	BLEQU	2\$...
	05	14	AC	D1	0002B	CMPL	20(AP), #5	4311
			13	13	0002F	BEQL	1\$...
		2A6C	C4	9F	00031	PUSHAB	P_AWR	4313
			01	DD	00035	PUSHL	#1	...
		00028362	8F	DD	00037	PUSHL	#164706	...
00000000G	00		03	FB	0003D	CALLS	#3, LIB\$SIGNAL	...
			05	DD	00044	PUSHL	#5	4315
	50		6442	D0	00046	MOVL	TERM_POINTER_TBL[R2], R0	4316
	51	FAAF	C4	9E	0004A	MOVAB	TABLEBASE, RT	...
			6140	9F	0004F	PUSHAB	(R1)[R0]	...
			7E	D4	00052	CLRL	-(SP)	4315
0000V	CF		03	FB	00054	CALLS	#3, DBG\$EXPRESSION_PARSER	...
			13	11	00059	BRB	3\$	4308
	50		6442	D0	0005B	MOVL	TERM_POINTER_TBL[R2], R0	4322
	51	FAAF	C4	9E	0005F	MOVAB	TABLEBASE, RT	...
			6140	9F	00064	PUSHAB	(R1)[R0]	...
			7E	D4	00067	CLRL	-(SP)	4321
0000V	CF		02	FB	00069	CALLS	#2, DBG\$EXPRESSION_PARSER	...
OC	BC		50	D0	0006E	MOVL	VALPTR, @VALUE_PTR	4323
	51		63	3C	00072	MOVZWL	(R3), R1	4330
	51	04	A3	C0	00075	ADDL2	4(R3), R1	...
	50		65	D0	00079	MOVL	CHARPTR, R0	...
	51		50	A3	0007C	SUBW3	R0, R1, (R3)	...
63	04		50	D0	00080	MOVL	R0, 4(R3)	4331
			60	91	00084	CMPB	(R0), #13	4332
	0D		04	12	00087	BNEQ	4\$...
	50		01	D0	00089	MOVL	#1, R0	4333
				04	0008C	RET		...
			50	D4	0008D	CLRL	R0	4335
			04	0008F	RET			...

; Routine Size: 144 bytes, Routine Base: DBG\$CODE + 0568

```

4223 4336 1 GLOBAL ROUTINE DBGSEXPRESSION_PARSER(ADDRESS_EXPRESSION, TERM_LIST) =
4224 4337 1
4225 4338 1 FUNCTION
4226 4339 1 This routine parses and interprets either a DEBUG Address Expression or
4227 4340 1 a language expression in the current language and returns the result of
4228 4341 1 the expression evaluation.
4229 4342 1
4230 4343 1 The routine uses an Operator Precedence parsing scheme. Each operator
4231 4344 1 is represented by an Operator Lexical Token Entry which contains the
4232 4345 1 kind of the operator (prefix, infix, or postfix) and the left and right
4233 4346 1 precedences of that operator. Operands are represented by Primary
4234 4347 1 Descriptors, Value Descriptors, or other Operand Lexical Token Entries.
4235 4348 1 The operators and operands are retrieved by calling the Primary Parser.
4236 4349 1
4237 4350 1 When an operand is encountered, it is simply stacked on the operand
4238 4351 1 stack. When an operator is encountered, its left precedence is com-
4239 4352 1 pared to the right precedence of the previous operator on the operator
4240 4353 1 stack. If the previous operator has the higher or equal precedence,
4241 4354 1 it is popped from the operator stack and evaluated. The evaluation
4242 4355 1 requires one or two operands to be popped from the operand stack, after
4243 4356 1 which the result is pushed back on that stack. When no previous opera-
4244 4357 1 tor has a higher or equal precedence, the new operator is pushed onto
4245 4358 1 the operator stack.
4246 4359 1
4247 4360 1 The operator stack is always initialized with the "initiator operator"
4248 4361 1 which ensures that there is always a previous operator on the stack.
4249 4362 1 The end of an expression is always signalled by the "terminator opera-
4250 4363 1 tor" whose left precedence is set such that it forces evaluation of all
4251 4364 1 operators still on the operator stack up to the initiator operator.
4252 4365 1 The single operand left on the operand stack thereafter constitutes
4253 4366 1 the result of the expression evaluation.
4254 4367 1
4255 4368 1 This routine accepts a list of allowed "terminator tokens" (keywords,
4256 4369 1 such as "DO" or "THEN" or special characters such as ":", ")", or "=",
4257 4370 1 depending on context). This list is passed to the Lexical Scanner
4258 4371 1 which returns the Terminator Operator when such a token or a carriage-
4259 4372 1 return is encountered. As a side effect, OWN variable TERMINATOR CODE
4260 4373 1 is set to a value which indicates which terminator token was found.
4261 4374 1 That terminator's character length is also set in TERMINATOR_LENGTH.
4262 4375 1 (This side effect is used when parsing subscript expressions.)
4263 4376 1
4264 4377 1 INPUTS
4265 4378 1 ADDRESS_EXPRESSION - A flag set to TRUE if a DEBUG Address Expression
4266 4379 1 is to be parsed and evaluated. If this flag is FALSE, a
4267 4380 1 language expression for the current language is parsed and
4268 4381 1 evaluated instead. This flag affects both the lexical
4269 4382 1 scanning of operator symbols and the parsing and evaluation
4270 4383 1 of the expression operators.
4271 4384 1
4272 4385 1 TERM_LIST - A vector of pointers to Terminator Lexical Token Entries
4273 4386 1 for the Terminator Tokens which can terminate the expression
4274 4387 1 to be parsed. The vector must be in PLIT form (TERM_LIST[-1]
4275 4388 1 gives the number of entries) and each pointer is expected to
4276 4389 1 be relative to TABLEBASE. If there are no terminator tokens
4277 4390 1 other than carriage return, this list is empty (0 entries).
4278 4391 1
4279 4392 1 3rd Optional Parameter - If this is present, and the value is

```


DBGSK DEPOSIT VERB then this is passed in from
DBGSEXP INT, from DBG\$NPARSE EXPRESSION, and from
DBG\$NPARSE DEPOSIT, so that in
DBG\$EXPRESSION_PARSE, when the expression is not
address expression and in deposit command,
DBG\$EVAL LANG OPERATOR will not be called with
DBG\$GL_IDENTITY_TOKEN.

OUTPUTS

A pointer to a Primary Descriptor or a Value Descriptor specifying the
result of the expression evaluation is returned as the
routine value.

BEGIN

MAP

TERM_LIST: REF VECTOR[,LONG]; ! Pointer to Terminator Table

LITERAL

MAX_OPAND_INDEX = 25; ! Maximum size of operand stack
MAX_OPTOR_INDEX = 25; ! Maximum size of operator stack

LOCAL

DEPOSIT_FLAG, ! A flag to indicate the call is
! from DBG\$NPARSE_DEPOSIT
JUNK, ! Output parameter - not used
LEFT_ARG, ! Pointer to the left (or only) argument
! of the current operator
LEFT_OP: REF TOKEN\$ENTRY, ! Pointer to top operator on operator
! stack--operator to the left
! of the current operator
NEW_PRIMPTR: REF DBG\$PRIMARY, ! Pointer to a Primary Descriptor
OP, ! The operator code for current operator
OPAND_INDEX, ! Current index into OPERAND_STACK
OPERAND_EXPECTED, ! Flag set when operand or prefix ope-
! rator is expected next
OPERAND_STACK: ! Parser's operand stack
! VECTOR[MAX_OPAND_INDEX],
OPERATOR_STACK: ! Parser's operator stack
! VECTOR[MAX_OPTOR_INDEX],
OPTOR_INDEX, ! Current index into OPERATOR_STACK
PAREN_NESTING, ! Parenthesis nesting depth in expr
PRIMPTR: REF DBG\$PRIMARY, ! Pointer to a Primary Descriptor
RIGHT_ARG, ! Pointer to the right argument to the
! current operator
TEMP_TOKEN: REF TOKEN\$ENTRY, ! Temporary copy of bit-select token
TOKEN: REF TOKEN\$ENTRY, ! Pointer to the current Token Entry
TOKEN_OPERAND_FLAG, ! Flag returned by Primary Parser
! saying an operand was returned
VALPTR: REF DBG\$PRIMARY; ! Pointer to Primary or Value Descriptor

BUILTIN ACTUALCOUNT,ACTUALPARAMETER;

DEPOSIT_FLAG = FALSE;


```
4337 4450 2 IF ACTUALCOUNT() GTR 2
4338 4451 THEN
4339 4452 BEGIN
4340 4453 IF ACTUALPARAMETER(3) NEQ DBG$K_DEPOSIT_VERB
4341 4454 THEN
4342 4455 $DBG_ERROR('DBGNPARSE\DBG$NPARSE_EXPRESSION');
4343 4456
4344 4457 DEPOSIT_FLAG = TRUE;
4345 4458 END;
4346 4459
4347 4460
4348 4461 ! Initialize the operator stack to contain one operator, the initiator
4349 4462 operator. Also initialize the operand stack to be empty. Indicate
4350 4463 that we expect an operand at the start of the parse.
4351 4464
4352 4465 OPTOR_INDEX = 0;
4353 4466 OPERATOR_STACK[OPTOR_INDEX] = INITIATOR_TOKEN;
4354 4467 OPAND_INDEX = -1;
4355 4468 OPERAND_EXPECTED = TRUE;
4356 4469 PAREN_NESTING = 0;
4357 4470
4358 4471
4359 4472 ! Loop through all operands and operators on the input line being parsed
4360 4473 until we reach the terminator operator. Stack operands and stack or
4361 4474 evaluate operators as appropriate until the terminator operator forces
4362 4475 evaluation of all stacked operators.
4363 4476
4364 4477 WHILE TRUE DO
4365 4478 BEGIN
4366 4479
4367 4480
4368 4481 ! Pick up the next operand or operator. An operand is returned as a
4369 4482 Primary Descriptor or a Value Descriptor by the Primary Parser.
4370 4483 An operator is just returned as an Operator Lexical Token Entry.
4371 4484
4372 4485 DBG$PRIMARY_PARSER(.OPERAND_EXPECTED, .ADDRESS_EXPRESSION,
4373 4486 .TERM_LIST, .PAREN_NESTING, TOKEN, TOKEN_OPERAND_FLAG);
4374 4487
4375 4488
4376 4489 ! Handle operands. If this is an operand, check that we are actually
4377 4490 expecting an operand at this point. Then stack the operand on the
4378 4491 operand stack and loop to get the next operand or operator.
4379 4492
4380 4493 IF .TOKEN_OPERAND_FLAG
4381 4494 THEN
4382 4495 BEGIN
4383 4496 IF NOT .OPERAND_EXPECTED
4384 4497 THEN
4385 4498 !<----- FIX UP MESSAGE -----
4386 4499 SIGNAL(DBG$_MISINVOPER, 1, UPLIT BYTE(%ASCIC 'somewhere'));
4387 4500
4388 4501 OPERAND_EXPECTED = FALSE;
4389 4502 OPAND_INDEX = .OPAND_INDEX + 1;
4390 4503 IF .OPAND_INDEX GEQ MAX_OPAND_INDEX THEN SIGNAL(DBG$_PARSTKOV);
4391 4504 OPERAND_STACK[.OPAND_INDEX] = .TOKEN;
4392 4505 IF .DBG$GL_DEVELOPER[3] THEN DUMP_PRIMARY(.TOKEN);
4393 4506 END
```

4394 4507 4
4395 4508 4
4396 4509 4
4397 4510 4
4398 4511 4
4399 4512 4
4400 4513 4
4401 4514 4
4402 4515 3
4403 4516 4
4404 4517 4
4405 4518 4
4406 4519 4
4407 4520 4
4408 4521 4
4409 4522 4
4410 4523 4
4411 4524 4
4412 4525 3
4413 4526 4
4414 4527 3
4415 4528 3
4416 4529 4
4417 4530 4
4418 4531 4
4419 4532 4
4420 4533 4
4421 4534 4
4422 4535 4
4423 4536 4
4424 4537 4
4425 4538 4
4426 4539 4
4427 4540 4
4428 4541 4
4429 4542 3
4430 4543 3
4431 4544 3
4432 4545 3
4433 4546 3
4434 4547 3
4435 4548 3
4436 4549 3
4437 4550 3
4438 4551 3
4439 4552 3
4440 4553 3
4441 4554 3
4442 4555 3
4443 4556 3
4444 4557 3
4445 4558 3
4446 4559 3
4447 4560 3
4448 4561 3
4449 4562 3
4450 4563 3

```
! Handle operators. First check that an operator of this operator's
! kind is expected. Then loop to pop all higher-precedence operators
! from the operator stack so they can be evaluated. When no higher-
! precedence operators are left on the stack, we stack the current
! operator on the operator stack.
ELSE
  BEGIN

    ! Check that an operator was expected unless this is a prefix
    ! operator (which is okay when we expect an operand). This check
    ! catches any expression which is not well-formed, including empty
    ! expressions. Also say that we expect an operand next unless
    ! this is a postfix operator.
    IF (.OPERAND_EXPECTED AND
        (.TOKEN[TOKEN$B_KIND] NEQ TOKEN$K_PREFIX_OP)) OR
        ((NOT .OPERAND_EXPECTED) AND
        (.TOKEN[TOKEN$B_KIND] EQL TOKEN$K_PREFIX_OP))
    THEN
      SIGNAL(DBG$_MISOPEMIS, 1, TOKEN[TOKEN$B_OPLEN]);

    IF .TOKEN[TOKEN$B_KIND] NEQ TOKEN$K_POSTFIX_OP
    THEN
      OPERAND_EXPECTED = TRUE;

    ! Pop and evaluate all operators on the operator stack whose prece-
    ! dence is higher than or equal to the current operator's. When we
    ! finally reach a lower-precedence operator, we exit the pop loop.
    WHILE TRUE DO
      BEGIN

        ! If the current operator (TOKEN) has a precedence higher than
        ! the precedence of the top operator on the stack (LEFT_OP),
        ! then leave this loop.
        LEFT_OP = .OPERATOR_STACK[.OPTOR_INDEX];
        IF .LEFT_OP[TOKEN$B_R_PREC] LSS .TOKEN[TOKEN$B_L_PREC]
        THEN
          EXITLOOP;

        ! The LEFT_OP operator has a higher or equal precedence. Pop
        ! it from the operator stack and evaluate it. Start by picking
        ! up its operands from the operand stack.
        OPTOR_INDEX = .OPTOR_INDEX - 1;
        IF .OPTOR_INDEX LSS 0
        THEN
          $DBG_ERROR('DBGPARSER\EXPRESSION_PARSER 1');
```

```
LEFT_ARG = .OPERAND_STACK[.OPAND_INDEX];
RIGHT_ARG = 0;
IF .LEFT_OP[TOKEN$B_KIND] EQL TOKEN$K_INFIX_OP
THEN
    BEGIN
        RIGHT_ARG = .LEFT_ARG;
        OPAND_INDEX = .OPAND_INDEX - 1;
        LEFT_ARG = .OPERAND_STACK[.OPAND_INDEX];
    END;

IF .DBG$GL_DEVELOPER[3] THEN DUMP_OPERATOR(.LEFT_OP, FALSE);

: If the operator we are about to evaluate is a "lexical"
: operator, meaning that it has some semantic effect on the
: lexical processing or parsing of the current expression,
: we perform that semantic action here.
IF .LEFT_OP[TOKEN$V_LEXICAL]
THEN
    BEGIN
        SELECTONE .LEFT_OP[TOKEN$W_CODE] OF
            SET

                : If evaluation of an open parenthesis operator is
                : forced by the terminator operator, we signal
                : unbalanced parentheses.
                [TOKEN$K_OPENPAREN]:
                    SIGNAL(DBG$UNBPAREN);

                : Restore the current radix to decimal. This undoes
                : a previous radix operator (such as %HEX) in the input.
                [TOKEN$K_RADIX_DEC]:
                    EXPRESSION_RADIX = DBG$K_DECIMAL;

                : Restore the current radix to hexadecimal. This undoes
                : a previous radix operator (such as %DEC) in the input.
                [TOKEN$K_RADIX_HEX]:
                    EXPRESSION_RADIX = DBG$K_HEX;

                : Restore the current radix to octal. This undoes a
                : previous radix operator (such as %HEX) in the input.
                [TOKEN$K_RADIX_OCT]:
                    EXPRESSION_RADIX = DBG$K_OCTAL;

                : Restore the current radix to binary. This undoes a
                : previous radix operator (such as %HEX) in the input.
```

```
4451 4564 5
4452 4565 5
4453 4566 5
4454 4567 5
4455 4568 6
4456 4569 6
4457 4570 6
4458 4571 6
4459 4572 5
4460 4573 5
4461 4574 5
4462 4575 5
4463 4576 5
4464 4577 5
4465 4578 5
4466 4579 5
4467 4580 5
4468 4581 5
4469 4582 5
4470 4583 5
4471 4584 6
4472 4585 6
4473 4586 6
4474 4587 6
4475 4588 6
4476 4589 6
4477 4590 6
4478 4591 6
4479 4592 6
4480 4593 6
4481 4594 6
4482 4595 6
4483 4596 6
4484 4597 6
4485 4598 6
4486 4599 6
4487 4600 6
4488 4601 6
4489 4602 6
4490 4603 6
4491 4604 6
4492 4605 6
4493 4606 6
4494 4607 6
4495 4608 6
4496 4609 6
4497 4610 6
4498 4611 6
4499 4612 6
4500 4613 6
4501 4614 6
4502 4615 6
4503 4616 6
4504 4617 6
4505 4618 6
4506 4619 6
4507 4620 6
```

```
4508 4621 6 [TOKEN$K_RADIX_BIN]:  
4509 4622 6     EXPRESSION_RADIX = DBG$K_BINARY;  
4510 4623 6  
4511 4624 6  
4512 4625 6     ! Infix NOT should not show up here.  
4513 4626 6  
4514 4627 6 [TOKEN$K_INFIX_NOT]:  
4515 4628 6     SIGNAL(DBG$MISOPEMIS, 1, LEFT_OP[TOKEN$B_OPLEN]);  
4516 4629 6  
4517 4630 6     ! Any other case is an error--a lexical operator should  
4518 4631 6     ! not be on the stack or should not be marked as lexical  
4519 4632 6     ! if it has no semantic action here.  
4520 4633 6  
4521 4634 6 [OTHERWISE]:  
4522 4635 6     $DBG_ERROR('DBGPARSER\EXPRESSION_PARSER 20');  
4523 4636 6  
4524 4637 6     TES;  
4525 4638 6  
4526 4639 6     END  
4527 4640 6  
4528 4641 6  
4529 4642 6     ! This is not a lexical operator--it is a 'normal' operator.  
4530 4643 6     ! If this is a DEBUG Address Expression, evaluate the Address  
4531 4644 6     ! Expression operator.  
4532 4645 6  
4533 4646 6 ELSE IF .ADDRESS_EXPRESSION  
4534 4647 5 THEN  
4535 4648 5     OPERAND_STACK[.OPAND_INDEX] =  
4536 4649 5     DBG$EVAL_ADDR_OPERATOR(.LEFT_OP, .LEFT_ARG, .RIGHT_ARG)  
4537 4650 5  
4538 4651 5  
4539 4652 5     ! And if it is a language expression, evaluate this non-lexical  
4540 4653 5     ! operator according to language rules.  
4541 4654 5  
4542 4655 5 ELSE  
4543 4656 6 BEGIN  
4544 4657 6  
4545 4658 6     ! EVAL_LANG_OPERATOR returns a pointer to a Primary Descriptor  
4546 4659 6     ! or a pointer to a Value Descriptor.  
4547 4660 6  
4548 4661 6     PRIMPTR = DBG$EVAL_LANG_OPERATOR(.LEFT_OP, .LEFT_ARG, .RIGHT_ARG);  
4549 4662 6  
4550 4663 6  
4551 4664 6     ! If a Primary Descriptor was returned from the expression  
4552 4665 6     ! evaluator then there may be some further processing to do.  
4553 4666 6     ! An example of this is the C expression:  
4554 4667 6     ! EVALUATE (*PTR).COMPONENT  
4555 4668 6     ! The (*PTR) will be evaluated by the expression evaluator  
4556 4669 6     ! and a Primary Descriptor will be returned (presumably,  
4557 4670 6     ! describing a record). Then we need to call the Primary  
4558 4671 6     ! Parser to pick up the rest of the expression.  
4559 4672 6  
4560 4673 6     IF .PRIMPTR[DBG$B_DHDR_TYPE] EQL DBG$K_PRIMARY_DESC  
4561 4674 6     THEN  
4562 4675 7         BEGIN  
4563 4676 7             DBG$PRIMARY_PARSER (  
4564 4677 7                 FALSE,                 ! Operand not expected
```



```
4565 4678 7 FALSE, | Not address expression
4566 4679 7 TERM_LIST, | Pass along terminator list
4567 4680 7 0 | Parenthesis nesting
4568 4681 7 NEW PRIMPTR, | Address to fill in result
4569 4682 7 JUNK, | Not used
4570 4683 7 PRIMPTR, | Input Primary
4571 4684 7 REMEMBER C_STATE GOT SUBSCRIPT);
4572 4685 7 PRIMPTR = .NEW_PRIMPTR;
4573 4686 6 END;
4574 4687 6
4575 4688 6
4576 4689 6 | Now just put the result of the expression on top of
4577 4690 6 the expression stack.
4578 4691 6
4579 4692 6 OPERAND_STACK[.OPAND_INDEX] = .PRIMPTR;
4580 4693 5 END;
4581 4694 4 END; | End of pop and evaluate loop
4582 4695 4
4583 4696 4
4584 4697 4
4585 4698 4 | If the new operator is a "lexical operator", meaning that it has
4586 4699 4 some semantic effect on the lexical processing or parsing of the
4587 4700 4 current expression, we perform that semantic action here. For
4588 4701 4 example, the terminator operator has the semantic effect of term-
4589 4702 4 inating the scan of the current expression.
4590 4703 4 IF .TOKEN[TOKEN$V_LEXICAL]
4591 4704 4 THEN
4592 4705 3 BEGIN
4593 4706 3 SELECTONE .TOKEN[TOKEN$W_CODE] OF
4594 4707 3 SET
4595 4708 3
4596 4709 3
4597 4710 3 | If this is the terminator operator, exit the parse loop.
4598 4711 3 |
4599 4712 3 [TOKEN$K_TERMINATOR]:
4600 4713 3 EXIT[LOOP];
4601 4714 3
4602 4715 3
4603 4716 3 | If this is an open parenthesis "(", increment the
4604 4717 3 parenthesis count.
4605 4718 3 |
4606 4719 3 [TOKEN$K_OPENPAREN]:
4607 4720 3 PAREN_NESTING = .PAREN_NESTING + 1;
4608 4721 3
4609 4722 3
4610 4723 3 | If this is a close parenthesis ")", decrement the paren-
4611 4724 3 thesis count, check for balanced parentheses, and remove
4612 4725 3 both the close parenthesis and the matching open paren-
4613 4726 3 thesis from the operator stack.
4614 4727 3 |
4615 4728 3 [TOKEN$K_CLOSEPAREN]:
4616 4729 3 BEGIN
4617 4730 3 PAREN_NESTING = .PAREN_NESTING - 1;
4618 4731 3 IF .PAREN_NESTING LSS 0 THEN SIGNAL(DBG$UNBPAREN);
4619 4732 3 IF .OPTR_INDEX LSS 1
4620 4733 3 THEN
4621 4734 3 $DBG_ERROR('DBGPARSER\EXPRESSION_PARSER 2');
```

```
.. 4622      4735      6
.. 4623      4736      6
.. 4624      4737      6
.. 4625      4738      6
.. 4626      4739      6
.. 4627      4740      6
.. 4628      4741      6
.. 4629      4742      6
.. 4630      4743      6
.. 4631      4744      6
.. 4632      4745      6
.. 4633      4746      6
.. 4634      4747      6
.. 4635      4748      6
.. 4636      4749      6
.. 4637      4750      6
.. 4638      4751      6
.. 4639      4752      6
.. 4640      4753      6
.. 4641      4754      6
.. 4642      4755      6
.. 4643      4756      6
.. 4644      4757      6
.. 4645      4758      6
.. 4646      4759      6
.. 4647      4760      6
.. 4648      4761      6
.. 4649      4762      6
.. 4650      4763      6
.. 4651      4764      6
.. 4652      4765      6
.. 4653      4766      6
.. 4654      4767      6
.. 4655      4768      6
.. 4656      4769      6
.. 4657      4770      6
.. 4658      4771      6
.. 4659      4772      6
.. 4660      4773      6
.. 4661      4774      6
.. 4662      4775      6
.. 4663      4776      6
.. 4664      4777      6
.. 4665      4778      6
.. 4666      4779      6
.. 4667      4780      6
.. 4668      4781      6
.. 4669      4782      6
.. 4670      4783      6
.. 4671      4784      6
.. 4672      4785      6
.. 4673      4786      6
.. 4674      4787      6
.. 4675      4788      6
.. 4676      4789      6
.. 4677      4790      6
.. 4678      4791      6
```

```
OPTOR_INDEX = .OPTOR_INDEX - 2;
TOKEN = .OPERATOR_STACK[.OPTOR_INDEX + 1];
END;
```

```
! If this is a set constant [1, 3..4, 10], call routine
! to build a value descriptor and push it on the operand
! stack. Ignore '[' by not pushing it on the operator
! stack.
```

```
[TOKENSK_OPENSET]:
```

```
  BEGIN
    OPERAND_EXPECTED = FALSE;
    OPAND_INDEX = .OPAND_INDEX + 1;
    IF .OPAND_INDEX GEQ MAX_OPAND_INDEX
    THEN
      SIGNAL(DBG$PARSTKOV);

    OPERAND_STACK[.OPAND_INDEX] = GET_SET_CONSTANT();
    IF .DBG$GL_DEVELOPER[3]
    THEN
      DUMP_PRIMARY(.OPERAND_STACK[.OPAND_INDEX]);

    TOKEN = .OPERATOR_STACK[.OPTOR_INDEX];
    OPTOR_INDEX = .OPTOR_INDEX - 1;
  END;
```

```
! If this is the bit-selection operator <pos,size,ext>
! allowed in Address Expressions and BLISS expressions,
! parse the operator's parameter fields (i.e., pick up
! pos, size, and ext) and incorporate those values in the
! Operator Lexical Token Entry put on the operator stack.
```

```
[TOKENSK_BITSELECT]:
```

```
  BEGIN
    TEMP_TOKEN = DBG$GET_TEMPMEM (
      TOKENSK_FIXED_SIZE_LONG +
      .TOKEN[TOKENSB_OPLEN]/4);
    CH$MOVE (
      TOKENSK_FIXED_SIZE_BYTE + .TOKEN[TOKENSB_OPLEN],
      .TOKEN, .TEMP_TOKEN);
    TOKEN = .TEMP_TOKEN;
    GET_FIELDREF(.TOKEN);
    TOKEN[TOKENSV_LEXICAL] = FALSE;
  END;
```

```
! Set the current radix to decimal. Here we put a lexical
! operator on the operator stack which will restore the
! current radix. We then set the radix to decimal.
```

```
[TOKENSK_RADIX_DEC]:
```

```
  BEGIN
    TOKEN = OPERATOR_TO_RESTORE_RADIX();
    EXPRESSION_RADIX = DBG$K_DECIMAL;
```

```
4679 4792 S
4680 4793 S
4681 4794 S
4682 4795 S
4683 4796 S
4684 4797 S
4685 4798 S
4686 4799 S
4687 4800 S
4688 4801 S
4689 4802 S
4690 4803 S
4691 4804 S
4692 4805 S
4693 4806 S
4694 4807 S
4695 4808 S
4696 4809 S
4697 4810 S
4698 4811 S
4699 4812 S
4700 4813 S
4701 4814 S
4702 4815 S
4703 4816 S
4704 4817 S
4705 4818 S
4706 4819 S
4707 4820 S
4708 4821 S
4709 4822 S
4710 4823 S
4711 4824 S
4712 4825 S
4713 4826 S
4714 4827 S
4715 4828 S
4716 4829 S
4717 4830 S
4718 4831 S
4719 4832 S
4720 4833 S
4721 4834 S
4722 4835 S
4723 4836 S
4724 4837 S
4725 4838 S
4726 4839 S
4727 4840 S
4728 4841 S
4729 4842 S
4730 4843 S
4731 4844 S
4732 4845 S
4733 4846 S
4734 4847 S
4735 4848 S
```

END;

```
! Set the current radix to hexadecimal. Here we put a lexical
! operator on the operator stack which will restore
! the current radix. We then set the radix to hexadecimal.
```

[TOKEN\$K_RADIX_HEX]:

BEGIN

TOKEN = OPERATOR_TO_RESTORE_RADIX();

EXPRESSION_RADIX = DBG\$K_HEX;

END;

```
! Set the current radix to octal. Here we put a lexical
! operator on the operator stack which will restore the
! current radix. We then set the radix to octal.
```

[TOKEN\$K_RADIX_OCT]:

BEGIN

TOKEN = OPERATOR_TO_RESTORE_RADIX();

EXPRESSION_RADIX = DBG\$K_OCTAL;

END;

```
! Set the current radix to binary. Here we put a lexical
! operator on the operator stack which will restore the
! current radix. We then set the radix to binary.
```

[TOKEN\$K_RADIX_BIN]:

BEGIN

TOKEN = OPERATOR_TO_RESTORE_RADIX();

EXPRESSION_RADIX = DBG\$K_BINARY;

END;

```
! Check for double-token (NOT =, NOT <, or NOT > in
! COBOL, where NOT is marked as infix and =, >, <
! is marked as prefix and lexical bit is set. Now
! at this point, we should have infix NOT on the
! operator stack, and have prefix token on hand.
! We'll replace the infix NOT on the operator stack
! to infix 'NOT <'.

```

[TOKEN\$K_PREFIX_EQL]:

BEGIN

TEMP_TOKEN = .OPERATOR_STACK[.OPTOR_INDEX];

IF .TEMP_TOKEN[TOKEN\$W_CODE] EQL TORENSK_INFIX_NOT

THEN

BEGIN

OPTOR_INDEX = .OPTOR_INDEX - 1;

TOKEN = COBOL_NOT_EQL_TOKEN;

END

ELSE

SIGNAL(DBG\$MISOPEMIS, 1, TOKEN[TOKEN\$B_OPLEN]);

END;

[illegible]


```
4793 4906 4
4794 4907 4
4795 4908 4
4796 4909 4
4797 4910 4
4798 4911 4
4799 4912 4
4800 4913 4
4801 4914 4
4802 4915 4
4803 4916 4
4804 4917 4
4805 4918 4
4806 4919 4
4807 4920 4
4808 4921 4
4809 4922 4
4810 4923 4
4811 4924 4
4812 4925 4
4813 4926 4
4814 4927 4
4815 4928 4
4816 4929 4
4817 4930 4
4818 4931 4
4819 4932 4
4820 4933 4
4821 4934 4
4822 4935 4
4823 4936 4
4824 4937 4
4825 4938 4
4826 4939 4
4827 4940 4
4828 4941 4
4829 4942 4
4830 4943 4
4831 4944 4
4832 4945 4

! Stack the current operator on the operator stack. Then loop to
! get the next operator or operand.
OPTOR_INDEX = .OPTOR_INDEX + 1;
IF .OPTOR_INDEX GEQ MAX_OPTOR_INDEX THEN SIGNAL(DBG$_PARSTKOVN);
OPERATOR_STACK[.OPTOR_INDEX] = .TOKEN;

END; ! End of ELSE-clause for operators

END; ! End of the get-symbol loop

! We are all done parsing the expression. Retrieve the descriptor from
! the top of the stack.
IF .OPAND_INDEX GTR 0 THEN $DBG_ERROR('DBGPARSER\EXPRESSION_PARSER 3');
VALPTR = .OPERAND_STACK[0];

! If this is a language expression, then we always return a Value Descriptor.
! Primary Descriptors or Volatile Value Descriptors are converted to
! Value Descriptors here. DBG$EVAL_LANG_OPERATOR does this for us.
IF NOT .ADDRESS_EXPRESSION AND
NOT .DEPOSIT_FLAG
THEN
VALPTR = DBG$EVAL_LANG_OPERATOR (DBG$GL_IDENTITY_TOKEN, .VALPTR, 0);

! If this is an address expression then we always return either a
! Primary Descriptor or a Volatile Value Descriptor.
! DBG$EVAL_ADDR_OPERATOR does this for us.
IF .ADDRESS_EXPRESSION
THEN
VALPTR = DBG$EVAL_ADDR_OPERATOR (DBG$GL_IDENTITY_TOKEN, .VALPTR, 0);

RETURN .VALPTR;
END;
```

```
.PSECT DBG$PLIT, NOWRT, SHR, PIC, 0

24 47 42 44 5C 45 53 52 41 50 4E 47 42 44 1F 03056 P.AWS: .ASCII <31>\DBGNPARSE\<92>\DBG$NPARSE_EXPRESS\
53 53 45 52 50 58 45 5F 45 53 52 41 50 4E 4E 03065
4E 4F 49 03073
52 50 58 45 5C 52 45 53 52 41 50 47 42 44 1D 03076 P.AWT: .ASCII \ION\
20 52 45 53 52 41 50 5F 4E 4F 49 53 53 45 03080 P.AWU: .ASCII <9>\somewhere\
52 50 58 45 5C 52 45 53 52 41 50 47 42 44 1E 0308F P.AWV: .ASCII <29>\DBGPARSER\<92>\EXPRESSION_PARSER \
20 52 45 53 52 41 50 5F 4E 4F 49 53 53 45 0309D
31 0309E
52 50 58 45 5C 52 45 53 52 41 50 47 42 44 1E 030AD P.AWV: .ASCII \1\
20 52 45 53 52 41 50 5F 4E 4F 49 53 53 45 030BD P.AWV: .ASCII <30>\DBGPARSER\<92>\EXPRESSION_PARSER \
52 50 58 45 5C 52 45 53 52 41 50 47 42 44 1D 030BB
20 52 45 53 52 41 50 5F 4E 4F 49 53 53 45 030CC P.AWV: .ASCII \20\
52 50 58 45 5C 52 45 53 52 41 50 47 42 44 1D 030BD P.AWV: .ASCII <29>\DBGPARSER\<92>\EXPRESSION_PARSER \
20 52 45 53 52 41 50 5F 4E 4F 49 53 53 45 030CC
```

```
52 50 58 45 5C 52 45 53 52 41 50 47 42 44 32 030DA
   20 52 45 53 52 41 50 5F 4E 4F 49 53 53 45 030DB
52 50 58 45 5C 52 45 53 52 41 50 47 42 44 33 030EA
   20 52 45 53 52 41 50 5F 4E 4F 49 53 53 45 030F8
   33 030F9
   45 03108
   33 03116
```

```
P.AWX: .ASCII \2\
        <29>\DBGPARSER\<92>\EXPRESSION_PARSER \
```

```
P.AWY: .ASCII \3\
        <29>\DBGPARSER\<92>\EXPRESSION_PARSER \
```

```
.PSECT DBG$CODE, NOWRT, SHR, PIC, 0
```

```
OFFC 00000
```

```
.ENTRY DBG$EXPRESSION_PARSER, Save R2,R3,R4,R5,R6,-, 4336
R7,R8,R9,R10,RT1
```

```
5E FF18 CE 9E 00002 MOVAB -232(SP), SP 4449
02 7E D4 00007 CLRL DEPOSIT_FLAG 4450
05 0C 6C 91 00009 CMPB (AP), #2 4453
00000000' 1E 1B 0000C BLEQU 2$ 4455
00028362 15 13 00012 BEQL 1$
00000000G 00 EF 9F 00014 PUSHAB P.AWS
6E 01 DD 0001A PUSHL #1
00028362 8F DD 0001C PUSHL #164706
00 03 FB 00022 CALLS #3, LIB$SIGNAL
6E 01 D0 00029 1$: MOVL #1, DEPOSIT_FLAG 4457
24 AE46 00000000' 56 D4 0002C 2$: CLRL OPTOR_INDEX 4465
04 57 01 CE 00037 MNEGL #1, OPAND_INDEX 4466
04 AE 01 D0 0003A MOVL #1, OPERAND_EXPECTED 4467
14 AE 9F 00040 3$: CLRL PAREN_NESTING 4468
1C AE 9F 00043 PUSHAB TOKEN_OPERAND_FLAG 4469
7E 04 5B DD 00046 PUSHAB TOKEN 4485
0000V CF 18 AC 7D 00048 MOVL PAREN_NESTING 4486
58 14 AE DD 0004C MOVQ ADDRESS_EXPRESSION, -(SP) 4485
46 04 AE DD 0004C PUSHL OPERAND_EXPECTED
15 00000000' 06 FB 0004F CALLS #6, DBG$PRIMARY_PARSER
58 18 AE D0 00054 MOVL TOKEN, R8 4504
46 14 AE E9 00058 BLBC TOKEN_OPERAND_FLAG, 6$ 4493
15 04 AE E8 0005C BLBS OPERAND_EXPECTED, 4$ 4496
00000000' EF 9F 00060 PUSHAB P.AWT 4499
000289AA 01 DD 00066 PUSHL #1
00000000G 00 8F DD 00068 PUSHL #166314
04 03 FB 0006E CALLS #3, LIB$SIGNAL
19 AE D4 00075 4$: CLRL OPERAND_EXPECTED 4501
57 D6 00078 INCL OPAND_INDEX 4502
000280E0 57 D1 0007A CMPL OPAND_INDEX, #25 4503
00 0D 19 0007D BLSS 5$
00000000G 00 8F DD 0007F PUSHL #164064
9C AD47 01 FB 00085 CALLS #1, LIB$SIGNAL
A7 00000000G 00 58 D0 0008C 5$: MOVL R8, OPERAND_STACK[OPAND_INDEX] 4504
00 03 E1 00091 BBC #3, DBG$GL_DEVELOPER, 3$ 4505
0000V CF 58 DD 00099 PUSHL R8
0A 04 01 FB 0009B CALLS #1, DUMP_PRIMARY
02 18 9E 11 000A0 BRB 3$ 4493
AE E9 000A2 6$: BLBC OPERAND_EXPECTED, 7$ 4525
BE 91 000A6 CMPB @TOKEN, #2 4526
```

			0A	12	000AA	BNEQ	8\$		
	1A	04	AE	E8	000AC	BLBS	OPERAND_EXPECTED, 9\$		4527
	02	18	BE	91	000B0	CMPB	3(TOKEN, -#2		4528
			14	12	000B4	BNEQ	9\$		
7E	1B	AE	0C	C1	000B6	ADDL3	#12, TOKEN, -(SP)		4530
			01	DD	000B8	PUSHL	#1		
		000289B2	8F	DD	000BD	PUSHL	#166322		
	00000000G	00	03	FB	000C3	CALLS	#3, LIB\$SIGNAL		
		04	68	91	000CA	CMPB	(R8), #4		4532
			04	13	000CD	BEQL	10\$		
	04	AE	01	D0	000CF	MOVL	#1, OPERAND_EXPECTED		4534
	59	24	AE	D0	000D3	MOVL	OPERATOR_STACK[OPOR_INDEX], LEFT_OP		4549
	05	A8	04	A9	91	CMPB	4(LEFT_OP), 5(R8)		4550
			03	1E	000DD	BGEQU	11\$		
			01	21	31	BRW	28\$		
	15		56	F4	000E2	SOBGEQ	OPOR_INDEX, 12\$		4559
		00000000'	EF	9F	000E5	PUSHAB	P,AWU		4562
			01	DD	000EB	PUSHL	#1		
		00028362	8F	DD	000ED	PUSHL	#164706		
	00000000G	00	03	FB	000F3	CALLS	#3, LIB\$SIGNAL		
	08	AE	9C	AD	47	MOVL	OPERAND_STACK[OPAND_INDEX], LEFT_ARG		4564
			0C	AE	D4	CLRL	RIGHT_ARG		4565
		03	69	91	00103	CMPB	(LEFT_OP), #3		4566
			0D	12	00106	BNEQ	13\$		
	0C	AE	08	AE	D0	MOVL	LEFT_ARG, RIGHT_ARG		4569
			57	D7	0010D	DECL	OPAND_INDEX		4570
	08	AE	9C	AD	47	MOVL	OPERAND_STACK[OPAND_INDEX], LEFT_ARG		4571
09	00000000G	00	03	E1	00115	BBC	#3, DBG\$GL_DEVELOPER, 14\$		4574
			7E	D4	0011D	CLRL	-(SP)		
			59	DD	0011F	PUSHL	LEFT_OP		
	0000V	CF	02	FB	00121	CALLS	#2, DUMP_OPERATOR		
7B		69	09	E1	00126	BBC	#9, (LEFT_OP), 24\$		4582
		0B	02	A9	B1	CMPW	2(LEFT_OP), #11		4593
			0F	12	0012E	BNEQ	16\$		
		000289FA	8F	DD	00130	PUSHL	#166394		4594
	00000000G	00	01	FB	00136	CALLS	#1, LIB\$SIGNAL		
			94	11	0013D	BRB	10\$		
		34	02	A9	B1	CMPW	2(LEFT_OP), #52		4600
			09	12	00143	BNEQ	18\$		
	00000000'	EF	0A	D0	00145	MOVL	#10, EXPRESSION_RADIX		4601
			85	11	0014C	BRB	10\$		
		35	02	A9	B1	CMPW	2(LEFT_OP), #53		4607
			09	12	00152	BNEQ	19\$		
	00000000'	EF	10	D0	00154	MOVL	#16, EXPRESSION_RADIX		4608
			60	11	0015B	BRB	25\$		
		36	02	A9	B1	CMPW	2(LEFT_OP), #54		4614
			09	12	00161	BNEQ	20\$		
	00000000'	EF	08	D0	00163	MOVL	#8, EXPRESSION_RADIX		4615
			51	11	0016A	BRB	25\$		
		37	02	A9	B1	CMPW	2(LEFT_OP), #55		4621
			09	12	00170	BNEQ	21\$		
	00000000'	EF	02	D0	00172	MOVL	#2, EXPRESSION_RADIX		4622
			42	11	00179	BRB	25\$		
		2C	02	A9	B1	CMPW	2(LEFT_OP), #44		4627
			0D	12	0017F	BNEQ	22\$		
			0C	A9	9F	PUSHAB	12(LEFT_OP)		4628
			01	DD	00184	PUSHL	#1		

			000289B2	8F	DD	00186	PUSHL	#166322		
				0E	11	0018C	BRB	23\$		
			00000000'	EF	9F	0018E	22\$: PUSHAB	P.AMV		4635
				01	DD	00194	PUSHL	#1		
			00028362	8F	DD	00196	PUSHL	#164706		
		00000000G	00	03	FB	0019C	23\$: CALLS	#3, LIB\$SIGNAL		
				98	11	001A3	BRB	15\$		4582
		16	04	AC	E9	001A5	24\$: BLBC	ADDRESS_EXPRESSION, 26\$		4649
			0C	AE	DD	001A9	PUSHL	RIGHT_ARG		
			0C	AE	DD	001AC	PUSHL	LEFT_ARG		
				59	DD	001AF	PUSHL	LEFT_OP		
		00000000G	00	03	FB	001B1	CALLS	#3, DBG\$EVAL_ADDR_OPERATOR		
			9C	AD	47	50	DO	001B8	MOV	RO, OPERAND_STACK[OPAND_INDEX]
				8D	11	001BD	25\$: BRB	17\$		4648
			0C	AE	DD	001BF	26\$: PUSHL	RIGHT_ARG		4661
			0C	AE	DD	001C2	PUSHL	LEFT_ARG		
				59	DD	001C5	PUSHL	LEFT_OP		
		00000000G	00	03	FB	001C7	CALLS	#3, DBG\$EVAL_LANG_OPERATOR		
			10	AE	DO	001CE	MOV	RO, PRIMPTR		
00000079	8F	10	BE	08	10	ED	001D2	CMPZV	#16, #8, @PRIMPTR, #121	4673
					1C	12	001DC	BNEQ	27\$	
					16	DD	001DE	PUSHL	#22	4676
			14	AE	DD	001E0	PUSHL	PRIMPTR		4683
			24	AE	9F	001E3	PUSHAB	JUNK		4676
			2C	AE	9F	001E6	PUSHAB	NEW PRIMPTR		
				7E	D4	001E9	CLRL	-(SP)		
			08	AC	DD	001EB	PUSHL	TERM_LIST		4679
				7E	7C	001EE	CLRL	-(SP)		4676
		0000V	CF	08	FB	001F0	CALLS	#8, DBG\$PRIMARY_PARSER		
			10	AE	DO	001F5	MOV	NEW PRIMPTR, PRIMPTR		4685
			9C	AD	47	10	AE	DO	001FA	27\$: MOV
					FED	031	00200	BRW	10\$	4692
					09	E1	00203	28\$: BBC	#9, (R8), 33\$	4541
					A8	B1	00207	CMPW	2(R8), #2	4703
					03	12	0020B	BNEQ	29\$	4712
					01BF	31	0020D	BRW	56\$	
			0B	02	A8	B1	00210	29\$: CMPW	2(R8), #11	4719
					04	12	00214	BNEQ	30\$	
					58	D6	00216	INCL	PAREN_NESTING	4720
					7A	11	00218	BRB	37\$	
			0C	02	A8	B1	0021A	30\$: CMPW	2(R8), #12	4728
					34	12	0021E	BNEQ	34\$	
			0D		5B	F4	00220	SOBGEQ	PAREN_NESTING, 31\$	4730
					8F	DD	00223	PUSHL	#166394	4731
		00000000G	00	01	FB	00229	CALLS	#1, LIB\$SIGNAL		
				56	D5	00230	31\$: TSTL	OPTOR_INDEX		4732
				15	14	00232	BGTR	32\$		
				EF	9F	00234	PUSHAB	P.AMV		4734
				01	DD	0023A	PUSHL	#1		
			00028362	8F	DD	0023C	PUSHL	#164706		
		00000000G	00	03	FB	00242	CALLS	#3, LIB\$SIGNAL		
				02	C2	00249	32\$: SUBL2	#2, OPTOR_INDEX		4736
				AE	46	DO	0024C	MOV	OPERATOR_STACK+4[OPTOR_INDEX], TOKEN	4737
			18		79	11	00252	33\$: BRB	39\$	4706
				39	02	A8	B1	00254	34\$: CMPW	2(R8), #57
					3C	12	00258	BNEQ	38\$	4746
				04	AE	D4	0025A	CLRL	OPERAND_EXPECTED	4748

	19		57	D6	0025D	INCL	OPAND_INDEX	4749		
			57	D1	0025F	CMPL	OPAND_INDEX, #25	4750		
		000280E0	0D	19	00262	BLSS	35\$			
			8F	DD	00264	PUSHL	#164064	4752		
00000000G	00		01	FB	0026A	CALLS	#1, LIBSSIGNAL			
0000V	CF		00	FB	00271	35\$:	#0, GET SET_CONSTANT	4754		
9C	AD47		50	D0	00276	MOVL	R0, OPERAND_STACK[OPAND_INDEX]			
09	00000000G	00	03	E1	0027B	BRB	#3, DBG\$GL_DEVELOPER, 38\$	4755		
			9C	AD47	DD	00283	OPERAND_STACK[OPAND_INDEX]	4757		
			01	FB	00287	CALLS	#1, DUMP_PRIMARY			
0000V	CF		24	AE46	D0	0028C	36\$:	4759		
18	AE		56	D7	00292	DECL	OPTOR_INDEX	4760		
			7F	11	00294	37\$:	4766			
			31	02	A8	B1	00296	38\$:	4770	
			33	12	0029A	BNEQ	40\$			
			50	0C	A8	9A	0029C		4774	
			50	04	C6	002A0	DIVL2	#4, R0		
				04	A0	9F	002A3	PUSHAB	4(R0)	4773
00000000G	00		01	FB	002A6	CALLS	#1, DBG\$GET_TEMPMEM			
	5A		50	D0	002AD	MOVL	R0, TEMP_TOKEN			
	50		0C	A8	9A	002B0	MOVZBL	12(R8), R0	4776	
	50		0D	C0	002B4	ADDL2	#13, R0			
6A	68		50	28	002B7	MOVC3	R0, (R8), (TEMP_TOKEN)	4777		
	18	AE	5A	D0	002BB	MOVL	TEMP_TOKEN, TOKEN	4778		
			18	AE	DD	002BF	PUSHL	TOKEN	4779	
0000V	CF		01	FB	002C2	CALLS	#1, GET_FIELDREF			
18	BE	0200	8F	AA	002C7	BICW2	#512, @TOKEN	4780		
			7D	11	002CD	39\$:	4786			
			34	02	A8	B1	002CF	40\$:	4788	
			12	12	002D3	BNEQ	41\$			
0000V	CF		00	FB	002D5	CALLS	#0, OPERATOR_TO_RESTORE_RADIX	4790		
18	AE		50	D0	002DA	MOVL	R0, TOKEN			
00000000'	EF		0A	D0	002DE	MOVL	#10, EXPRESSION_RADIX	4791		
			65	11	002E5	BRB	46\$	4796		
			35	02	A8	B1	002E7	41\$:	4799	
			12	12	002EB	BNEQ	42\$			
0000V	CF		00	FB	002ED	CALLS	#0, OPERATOR_TO_RESTORE_RADIX	4801		
18	AE		50	D0	002F2	MOVL	R0, TOKEN			
00000000'	EF		10	D0	002F6	MOVL	#16, EXPRESSION_RADIX	4802		
			6A	11	002FD	BRB	48\$	4706		
			36	02	A8	B1	002FF	42\$:	4810	
			12	12	00303	BNEQ	44\$			
0000V	CF		00	FB	00305	CALLS	#0, OPERATOR_TO_RESTORE_RADIX	4812		
18	AE		50	D0	0030A	MOVL	R0, TOKEN			
00000000'	EF		08	D0	0030E	MOVL	#8, EXPRESSION_RADIX	4813		
			71	11	00315	BRB	50\$	4706		
			37	02	A8	B1	00317	44\$:	4821	
			12	12	0031B	BNEQ	45\$			
0000V	CF		00	FB	0031D	CALLS	#0, OPERATOR_TO_RESTORE_RADIX	4823		
18	AE		50	D0	00322	MOVL	R0, TOKEN			
00000000'	EF		02	D0	00326	MOVL	#2, EXPRESSION_RADIX	4824		
			59	11	0032D	BRB	50\$	4706		
			0041	8F	02	A8	B1	0032F	45\$:	4836
			17	12	00335	BNEQ	47\$			
	5A		24	AE46	D0	00337	MOVL	OPERATOR_STACK[OPTOR_INDEX], TEMP_TOKEN	4838	
	2C		02	AA	B1	0033C	CMPL	2(TEMP_TOKEN), #44	4839	
			48	12	00340	BNEQ	51\$			

18	AE	00000000'	56	D7	00342	DECL	OPTOR_INDEX	4842	
			EF	9E	00344	MOVAB	COBOL_NOT_EQL_TOKEN, TOKEN	4843	
	3F	02	64	11	0034C	BRB	54\$	4839	
			A8	B1	0034E	CMPW	2(R8), #63	4853	
	5A	24	17	12	00352	BNEQ	49\$		
	2C	02	AE46	D0	00354	MOVL	OPERATOR_STACK[OPTOR_INDEX], TEMP_TOKEN	4855	
			AA	B1	00359	CMPW	2(TEMP_TOKEN), #44	4856	
			2B	12	0035D	BNEQ	51\$		
			56	D7	0035F	DECL	OPTOR_INDEX	4859	
18	AE	00000000'	EF	9E	00361	MOVAB	COBOL_NOT_GTR_TOKEN, TOKEN	4860	
			47	11	00369	BRB	54\$	4856	
0040	8F	02	A8	B1	0036B	CMPW	2(R8), #64	4870	
			24	12	00371	BNEQ	52\$		
	5A	24	AE46	D0	00373	MOVL	OPERATOR_STACK[OPTOR_INDEX], TEMP_TOKEN	4872	
	2C	02	AA	B1	00378	CMPW	2(TEMP_TOKEN), #44	4873	
			0C	12	0037C	BNEQ	51\$		
			56	D7	0037E	DECL	OPTOR_INDEX	4876	
18	AE	00000000'	EF	9E	00380	MOVAB	COBOL_NOT_LSS_TOKEN, TOKEN	4877	
			28	11	00388	BRB	54\$	4873	
		0C	A8	9F	0038A	PUSHAB	12(R8)	4881	
			01	DD	0038D	PUSHL	#1		
		000289B2	8F	DD	0038F	PUSHL	#166322		
			14	11	00395	BRB	53\$		
	2C	02	A8	B1	00397	CMPW	2(R8), #44	4888	
			15	13	0039B	BEQL	54\$		
		00000000'	EF	9F	0039D	PUSHAB	P.AWX	4899	
			01	DD	003A3	PUSHL	#1		
		00028362	8F	DD	003A5	PUSHL	#164706		
00000000G	00		03	FB	003AB	CALLS	#3, LIB\$SIGNAL		
			56	D6	003B2	INCL	OPTOR_INDEX	4909	
	19		56	D1	003B4	CMPW	OPTOR_INDEX, #25	4910	
			0D	19	003B7	BLSS	55\$		
		000280E0	8F	DD	003B9	PUSHL	#164064		
00000000G	00		01	FB	003BF	CALLS	#1, LIB\$SIGNAL		
	24	AE46	18	AE	D0	003C6	MOVL	TOKEN, OPERATOR_STACK[OPTOR_INDEX]	4911
			FC71	31	003CC	BRW	3\$	4477	
			57	D5	003CF	TSTL	OPAND_INDEX	4921	
			15	15	003D1	BLEQ	57\$		
		00000000'	EF	9F	003D3	PUSHAB	P.AWY		
			01	DD	003D9	PUSHL	#1		
		00028362	8F	DD	003DB	PUSHL	#164706		
00000000G	00		03	FB	003E1	CALLS	#3, LIB\$SIGNAL		
	50	9C	AD	D0	003E8	MOVL	OPERAND_STACK, VALPTR	4922	
	18	04	AC	E8	003EC	BLBS	ADDRESS_EXPRESSION, 59\$	4929	
	11		6E	E8	003F0	BLBS	DEPOSIT_FLAG, 58\$	4930	
			7E	D4	003F3	CLRL	-(SP)	4932	
			50	DD	003F5	PUSHL	VALPTR		
		00000000'	EF	9F	003F7	PUSHAB	DBG\$GL_IDENTITY_TOKEN		
00000000G	00		03	FB	003FD	CALLS	#3, DBG\$EVAL_LANG_OPERATOR		
	11	04	AC	E9	00404	BLBC	ADDRESS_EXPRESSION, 60\$	4939	
			7E	D4	00408	CLRL	-(SP)	4941	
			50	DD	0040A	PUSHL	VALPTR		
		00000000'	EF	9F	0040C	PUSHAB	DBG\$GL_IDENTITY_TOKEN		
00000000G	00		03	FB	00412	CALLS	#3, DBG\$EVAL_ADDR_OPERATOR		
			04	00419	60\$:	RET		4945	

DBGPARSER
V04-000

M 10
16-Sep-1984 02:10:13
14-Sep-1984 12:17:30

VAX-11 B11ss-32 V4.0-742
[DEBUG.SRC]DBGPARSER.B32;1

Page 170
(21)

```
4834 4946 1 GLOBAL ROUTINE DBG$GET_BIF_ARGUMENTS (LENGTH, NAME) =
4835 4947 1
4836 4948 1 FUNCTION
4837 4949 1     This routine picks up built-in function arguments. It calls
4838 4950 1     DBG$EXPRESSION_PARSER to parse and evaluate each argument
4839 4951 1     expression.
4840 4952 1
4841 4953 1     This routine assumes that the opening set parenthesis has
4842 4954 1     already been found and that the parse pointer points to the
4843 4955 1     start of the first built-in function argument expression.
4844 4956 1     When this routine returns, the parse pointer is left pointing
4845 4957 1     at the first character after the closing set parenthesis.
4846 4958 1
4847 4959 1 INPUTS
4848 4960 1     LENGTH - The number of arguments expected.
4849 4961 1     NAME    - Name of Built-in function or Ada tick operator to output
4850 4962 1               when an error is to be signaled.
4851 4963 1
4852 4964 1 OUTPUTS
4853 4965 1     Pointer to a counted vector of long words that point to the
4854 4966 1     argument values. The vector will have a minimum length
4855 4967 1     of 2 arguments.
4856 4968 1
4857 4969 1
4858 4970 1 BEGIN
4859 4971 1
4860 4972 1 MAP
4861 4973 1     NAME      : REF VECTOR [,BYTE];           ! Name of function or tick operator
4862 4974 1
4863 4975 1 LOCAL
4864 4976 1     ARG_PTR    : REF VECTOR [,LONG],           ! Pointer to counted vect. of arguments
4865 4977 1     BIF_INDEX; ! Index into the vector
4866 4978 1
4867 4979 1
4868 4980 1 ! Get temporary memory for the argument list.
4869 4981 1
4870 4982 1 ARG_PTR = DBG$GET_TEMPMEM( MAX((.LENGTH + 1), 3) );
4871 4983 1
4872 4984 1 ! Set the vector index to i and loop, picking up arguments, until
4873 4985 1 ! a closing paren is found. The arguments are picked up by a
4874 4986 1 ! recursive call on DBG$EXPRESSION_PARSER.
4875 4987 1
4876 4988 1 BIF_INDEX = 1;
4877 4989 1 TERMINATOR_CODE = TOKEN$K_TERM_NONE;
4878 4990 1 WHILE .TERMINATOR_CODE NEQ TOKEN$K_TERM_CLOSE DO
4879 4991 1     BEGIN
4880 4992 1
4881 4993 1         ! If the number of arguments found so far is more than the number
4882 4994 1         ! expected, signal a bad argument list.
4883 4995 1
4884 4996 1         IF .BIF_INDEX GTR .LENGTH
4885 4997 1         THEN
4886 4998 1             SIGNAL(DBG$INVARGLIS, 1, .NAME);
4887 4999 1
4888 5000 1         ! Recursive call to pick up the argument, and store the pointer
4889 5001 1         ! away in the vector.
4890 5002 1
```



```
4891 5003 ARG_PTR[.BIF_INDEX] = DBG$EXPRESSION_PARSER(FALSE, COMPAREN_TERM_TBL);
4892 5004
4893 5005 : If an invalid terminator was found signal the error. If not,
4894 5006 : increment the character pointer past the terminator.
4895 5007
4896 5008 IF .TERMINATOR_CODE EQL TOKEN$K_TERM_NONE
4897 5009 THEN
4898 5010 SIGNAL(DBG$MISCLOSUB);
4899 5011 CHARPTR = .CHARPTR + .TERMINATOR_LENGTH;
4900 5012 BIF_INDEX = .BIF_INDEX + 1;
4901 5013
4902 5014 END;
4903 5015
4904 5016 BIF_INDEX = .BIF_INDEX - 1;
4905 5017
4906 5018 : If the number of arguments found is not the same as the number of
4907 5019 : expected, signal a bad argument list.
4908 5020
4909 5021 IF .BIF_INDEX NEQ .LENGTH
4910 5022 THEN
4911 5023 SIGNAL(DBG$INVARGLIS, 1, .NAME);
4912 5024
4913 5025 : If only one argument is requested, a zero is left for the second argument.
4914 5026 : This is done for the subsequent call to DBG$EVAL_LANG_OPERATOR.
4915 5027 : Then store away the number of arguments and return a vector of these
4916 5028 : arguments.
4917 5029
4918 5030 ARG_PTR[0] = .BIF_INDEX;
4919 5031
4920 5032 RETURN .ARG_PTR;
4921 5033 END;
```

50	04	AC	00000000G	003C	00000	.ENTRY	DBG\$GET BIF_ARGUMENTS, Save R2,R3,R4,R5	4946
			54 00000000'	00 9E	00002	MOVAB	LIB\$SIGNAL, R5	
				EF 9E	00009	MOVAB	TERMINATOR_CODE, R4	
				01 C1	00010	ADDL3	#1, LENGTH, R0	4982
				50 DD	00015	PUSHL	R0	
		03		6E D1	00017	CMPL	(SP), #3	
				03 18	0001A	BGEQ	1\$	
		6E		03 D0	0001C	MOVL	#3, (SP)	
		00000000G	00	01 FB	0001F 1\$:	CALLS	#1, DBG\$GET_TEMP MEM	
			53	50 D0	00026	MOVL	R0, ARG_PTR	
			52	01 D0	00029	MOVL	#1, BIF_INDEX	4988
				64 D4	0002C	CLRL	TERMINATOR_CODE	4989
		02		64 D1	0002E 2\$:	CMPL	TERMINATOR_CODE, #2	4990
				3C 13	00031	BEQL	5\$	
		04	AC	52 D1	00033	CMPL	BIF_INDEX, LENGTH	4996
				0E 15	00037	BLEQ	3\$	
			08	AC DD	00039	PUSHL	NAME	4998
				01 DD	0003C	PUSHL	#1	
			00028838	8F DD	0003E	PUSHL	#165944	
		65		03 FB	00044	CALLS	#3, LIB\$SIGNAL	
			00000000'	EF 9F	00047 3\$:	PUSHAB	COMPAREN_TERM_TBL	5003

FB92	CF		7E	D4	0004D	CLRL	-(SP)	
	6342		02	FB	0004F	CALLS	#2, DBG\$EXPRESSION_PARSER	
			50	D0	00054	MOVL	R0, (ARG_PTR)[BIF_INDEX]	
			64	D5	00058	TSTL	TERMINATOR_CODE	5008
			09	12	0005A	BNEQ	4\$	
		00028E90	8F	DD	0005C	PUSHL	#167568	5010
FBCC	65		01	FB	00062	CALLS	#1, LIB\$SIGNAL	
	C4	04	A4	C0	00065	ADDL2	TERMINATOR_LENGTH, CHARPTR	5011
			32	D6	0006B	INCL	BIF_INDEX	5012
			BF	11	0006D	BRB	2\$	4990
			52	D7	0006F	DECL	BIF_INDEX	5016
04	AC		52	D1	00071	CMPL	BIF_INDEX, LENGTH	5021
			0E	13	00075	BEQL	6\$	
		08	AC	DD	00077	PUSHL	NAME	5023
			01	DD	0007A	PUSHL	#1	
		00028838	8F	DD	0007C	PUSHL	#165944	
	65		03	FB	00082	CALLS	#3, LIB\$SIGNAL	
	63		52	D0	00085	MOVL	BIF_INDEX, (ARG_PTR)	5030
	50		53	D0	00088	MOVL	ARG_PTR, R0	5032
			04	0008B	RET			5033

; Routine Size: 140 bytes, Routine Base: DBG\$CODE + 0A12

```
4923 5034 1 GLOBAL ROUTINE DBGSLEXICAL_SCANNER(OPERAND_EXPECTED,  
4924 5035 1 ADDRESS_EXPRESSION, TERM_LIST, PAREN_NESTING) =  
4925 5036 1  
4926 5037 1 FUNCTION  
4927 5038 1 This routine is the Lexical Scanner used during expression parsing.  
4928 5039 1 It scans the character or characters pointed to by CHARPTR to pick  
4929 5040 1 up the next lexical token according to the rules of the current  
4930 5041 1 language. It picks up or constructs a Lexical Token Entry for the  
4931 5042 1 found lexical token and returns a pointer to that Token Entry as its  
4932 5043 1 result.  
4933 5044 1  
4934 5045 1 The Lexical Scanner is called by the Primary Parser which then uses  
4935 5046 1 the returned token to build up a Primary Descriptor if the token is  
4936 5047 1 part of a Primary Symbol. If the returned token is not part of a  
4937 5048 1 Primary Symbol (i.e., if it is an operator in the current language or  
4938 5049 1 the current Address Expression or if it is a constant), the Primary  
4939 5050 1 Parser returns it directly to the Expression Parser. The Expression  
4940 5051 1 Parser is thus fed a stream of operands and operators which it then  
4941 5052 1 uses to interpret and evaluate the current expression.  
4942 5053 1  
4943 5054 1 This routine assumes that the input line being scanned has already  
4944 5055 1 been converted to upper case and it assumes that the input line is  
4945 5056 1 terminated by a carriage-return character. It also assumes that  
4946 5057 1 the variable CHARPTR has been set up to point to the current posi-  
4947 5058 1 tion in the buffer being scanned. CHARPTR is updated by this rou-  
4948 5059 1 tine to point to the first character position after the current  
4949 5060 1 token.  
4950 5061 1  
4951 5062 1 This routine accepts a list of allowed "terminator tokens" (keywords  
4952 5063 1 such as "DO" or "THEN" or special characters such as "(", ")", or "=",  
4953 5064 1 depending on context). This list is passed to the Lexical Scanner  
4954 5065 1 which returns the Terminator Operator when such a token or a carriage-  
4955 5066 1 return is encountered. As a side effect, OWN variable TERMINATOR_CODE  
4956 5067 1 is set to a value which indicates which terminator token was found.  
4957 5068 1 That terminator's character length is also set in TERMINATOR_LENGTH.  
4958 5069 1 (This side effect is used when parsing subscript expressions.)  
4959 5070 1  
4960 5071 1 INPUTS  
4961 5072 1 OPERAND_EXPECTED - A flag set to TRUE if an operand is expected next  
4962 5073 1 in the parse of the current expression. This flag is used  
4963 5074 1 to disambiguate certain operators, such as "+" and "-",  
4964 5075 1 which are prefix operators when an operand is expected  
4965 5076 1 next and are infix operators when an operator is expected  
4966 5077 1 next.  
4967 5078 1  
4968 5079 1 ADDRESS_EXPRESSION - A flag set to TRUE if we are parsing a DEBUG  
4969 5080 1 Address Expression instead of a language expression. This  
4970 5081 1 affects the parsing of Address Expression operators such  
4971 5082 1 as "++", "--", "/", and "a" which are recognized by  
4972 5083 1 DEBUG rules, not language rules, in Address Expressions.  
4973 5084 1  
4974 5085 1 TERM_LIST - A vector of pointers to Terminator Lexical Token Entries  
4975 5086 1 for the Terminator Tokens which can terminate the expression  
4976 5087 1 to be parsed. The vector must be in PLIT form (TERM_LIST[-1]  
4977 5088 1 gives the number of entries) and each pointer is expected to  
4978 5089 1 be relative to TABLEBASE. If there are no terminator tokens  
4979 5090 1 other than carriage return, this list is empty (0 entries).
```

```
4980 5091 1
4981 5092 1
4982 5093 1
4983 5094 1
4984 5095 1
4985 5096 1
4986 5097 1
4987 5098 1
4988 5099 1
4989 5100 1
4990 5101 1
4991 5102 1
4992 5103 1
4993 5104 1
4994 5105 2
4995 5106 2
4996 5107 2
4997 5108 2
4998 5109 2
4999 5110 2
5000 5111 2
5001 5112 2
5002 5113 2
5003 5114 2
5004 5115 2
5005 5116 2
5006 5117 2
5007 5118 2
5008 5119 2
5009 5120 2
5010 5121 2
5011 5122 2
5012 5123 2
5013 5124 2
5014 5125 2
5015 5126 2
5016 5127 2
5017 5128 2
5018 5129 2
5019 5130 2
5020 5131 2
5021 5132 2
5022 5133 2
5023 5134 2
5024 5135 2
5025 5136 2
5026 5137 2
5027 5138 2
5028 5139 2
5029 5140 2
5030 5141 2
5031 5142 2
5032 5143 2
5033 5144 2
5034 5145 2
5035 5146 2
5036 5147 2
```

PAREN_NESTING - The current parenthesis nesting depth. This parameter is used to detect whether certain tokens are expression terminators or not. (For example, a ")" token in a subscript expression terminates it only if parentheses are already balanced.)

OUTPUTS
A pointer to the Lexical Token Entry for the next lexical token found in the input line is returned as the routine value. If there are no more lexical tokens on the line, a pointer to a Token Entry for the TOKEN\$K_TERMINATOR operator is returned.

BEGIN

MAP
TERM_LIST: REF VECTOR[,LONG]; : Pointer to Terminator Table to use

LABEL
CHECK_THIS_TERMINATOR; : Label used to leave terminator checking code

LOCAL
ACTION, : Action index during number scanning
BACKUP_DIGIT_PTR, : Pointer to last good digit in number
: --used to back up number scan
BACKUP_NUMBER_KIND, : Kind of numeric constant definite so far--used to back up number scan
BEST_TOKEN_FOUND, : Pointer to the Operator Token Entry for an operator with the right name but the wrong kind
CLASS, : Character class code of current character during number scanning
ENDPTR, : Pointer to last char in an identifier
ERRORMSG, : Error message condition code
INDEX, : Index into look-up tables
NAMEPTR: REF VECTOR[,BYTE], : Pointer to name string in Percent Tbl
NEW_STARTPTR, : Start pointer to lower case identifier
NUMBER_KIND, : Kind of Numeric Constant Token found
PRID: REF PRID\$ENTRY, : Pointer to Predefined Identifier Entry
QUOTE, : Quote character which started the current quoted string constant
STARTPTR: REF VECTOR[,BYTE], : Pointer to start of current token
STATE_INDEX, : Current Number Scanner State Table index during number scanning
TERMPTR: REF TOKEN\$ENTRY, : Pointer to Terminator Token Entry
TOKEN: REF TOKEN\$ENTRY, : Pointer to Operator Token Entry
TOKENBUFFER: VECTOR[256,BYTE], : Vector in which current token is accumulated as Counted ASCII
TOKEN_TYPE, : Token's type; eg. TOKEN\$K_STRING
TOKENLEN; : The character length of current token

: Start by scanning past any leading blanks. Then mark the start location
: of the token to be picked up.


```
!
WHILE .CHARTBL[.CHARPTR[0], CHRTBL$V_SPACE] DO
    CHARPTR = .CHARPTR + 1;

STARTPTR = .CHARPTR;

! If this took us to the carriage-return, return the terminator operator.
!
IF .CHARTBL[0] EQL CAR_RET
THEN
    BEGIN
        TERMINATOR_CODE = TOKEN$K_TERM_NONE;
        TERMINATOR_LENGTH = 0;
        RETURN TERMINATOR_TOKEN;
    END;

! Check for a valid Terminator Token. The TERM_LIST parameter lists all
! valid terminator tokens for the current expression. If we have one of
! those here, we set TERMINATOR_CODE appropriately and return the termi-
! nator operator.
IF .CHARTBL[.CHARPTR[0], CHRTBL$V_TERMINATOR]
THEN
    BEGIN

        ! Loop over all allowed terminators in this context.
        !
        INCR I FROM 0 TO .TERM_LIST[-1] - 1 DO
            BEGIN
                TERMPTR = .TERM_LIST[I] + TABLEBASE;

                ! Check whether this terminator matches what we have in the input
                ! line being scanned.
                !
                CHECK_THIS_TERMINATOR:
                BEGIN
                    IF CH$NEQ(.TERMPTR[TOKEN$B_LENGTH], TERMPTR[TOKEN$A_NAME],
                        .TERMPTR[TOKEN$B_LENGTH], .CHARPTR, 0)
                    THEN
                        LEAVE CHECK_THIS_TERMINATOR;

                    IF .CHARTBL[.CHARPTR[0], CHRTBL$V_ALPHABETIC] AND
                        (.CHARTBL[.CHARPTR[.TERMPTR[TOKEN$B_LENGTH]],
                            CHRTBL$V_IDENT_MIDDLE] OR
                            .CHARTBL[.CHARPTR[.TERMPTR[TOKEN$B_LENGTH]],
                                CHRTBL$V_IDENT_END])
                    THEN
                        LEAVE CHECK_THIS_TERMINATOR;

                    IF .TERMPTR[TOKEN$V_BALANCED_PARENS] AND (.PAREN_NESTING NEQ 0)
                    THEN
                        LEAVE CHECK_THIS_TERMINATOR;
```

```
5094      IF .TERMPTR[TOKEN$V_MUST BE SINGLE] AND
5095      (.CHARPTR[0] EQL .CHARPTR[1])
5096      THEN
5097      LEAVE CHECK_THIS_TERMINATOR;
5098
5099      ! This is a valid terminator in this context. As a side-effect
5100      ! save the terminator code in TERMINATOR_CODE and its length in
5101      ! TERMINATOR_LENGTH. Then return the terminator operator.
5102      TERMINATOR_CODE = .TERMPTR[TOKEN$V_CODE];
5103      TERMINATOR_LENGTH = .TERMPTR[TOKEN$V_LENGTH];
5104      RETURN TERMINATOR_TOKEN;
5105
5106      END;
5107      ! End of CHECK_THIS_TERMINATOR block
5108
5109      END;
5110      ! End of INCR loop over terminators
5111
5112      END;
5113      ! End of terminator checking
5114
5115      ! Handle any language-specific special cases that must be sorted out before
5116      ! we go through the normal lexical scanning code below. Here we check for
5117      ! those tokens that would be scanned incorrectly if we went through the
5118      ! normal scanning mechanisms below.
5119
5120      IF .CHARTBL[.CHARPTR[0], CHRTBL$V_SPECIAL_CASE]
5121      THEN
5122      BEGIN
5123      CASE .DBG$GB_LANGUAGE FROM DBG$K_MIN_LANGUAGE TO DBG$K_MAX_LANGUAGE OF
5124      SET
5125
5126      ! Handle PL/I. Here we special-case the -> operator. We need to
5127      ! mark "-" as being a single-character operator in the character
5128      ! table so it does not get combined with itself or other operator
5129      ! characters in expressions. Hence we must special-case the one
5130      ! situation where it can be combined, namely in "->".
5131      [DBG$K_PLI]:
5132      BEGIN
5133      IF (.CHARPTR[0] EQL '-') AND (.CHARPTR[1] EQL '>')
5134      THEN
5135      BEGIN
5136      CHARPTR = .CHARPTR + 2;
5137      RETURN PLI_ARROW_TOKEN;
5138      END;
5139
5140      END;
5141
5142      ! There are no special cases for any other language. If we get
5143      ! here, something is wrong (the code or character table is wrong.)
5144
5145      [INRANGE, OVRANGE]:
5146      SDBG_ERROR('DBGPARSER\LEXICAL_SCANNER 10');
```

5151
5152
5153
5154
5155
5156
5157
5158
5159
5160
5161
5162
5163
5164
5165
5166
5167
5168
5169
5170
5171
5172
5173
5174
5175
5176
5177
5178
5179
5180
5181
5182
5183
5184
5185
5186
5187
5188
5189
5190
5191
5192
5193
5194
5195
5196
5197
5198
5199
5200
5201
5202
5203
5204
5205
5206
5207

TES;
END;

! If the next token is one of the "special" symbols ".", "\", or "'",
! and an operand is expected here, see if this token
! could mean "current location", "current value", or "previous
! location". If so, return the appropriate Identifier Lexical Token Entry
! but change the name to %CURLOC, %CURVAL, or %PREVLOC.

IF .OPERAND_EXPECTED AND .CHARTBL[.CHARPTR[0], CHRTBL\$V_SPECIAL_SYMBOL]
THEN

BEGIN
CHARPTR = .CHARPTR + 1;
WHILE .CHARTBL[.CHARPTR[0], CHRTBL\$V_SPACE] DO
CHARPTR = .CHARPTR + 1;

! Handle the special case where the user has said
! SET BREAK . DO (command-list). In this case we assume that
! the "." means "current location".

IF .DBG\$GB_SET_BREAK_FLAG
THEN

BEGIN
IF (.CHARPTR[0] EQL 'D') AND
(.CHARPTR[1] EQL 'O') AND
((.CHARPTR[2] EQL ' ') OR (.CHARPTR[2] EQL '('))
THEN
RETURN CURLOC_TOKEN;
IF (.CHARPTR[0] EQL 'W') AND
(.CHARPTR[1] EQL 'H') AND
(.CHARPTR[2] EQL 'E') AND
(.CHARPTR[3] EQL 'N') AND
((.CHARPTR[4] EQL ' ') OR (.CHARPTR[4] EQL '('))
THEN
RETURN CURLOC_TOKEN;
END;

! We need to determine whether this special symbol is really one
! of the debugger permanent symbols for previous, current, or next
! location, or whether it is a language operator.

For "\", the only other possible meaning is as the start of a pathname.
(not the middle, since then "OPERAND EXPECTED" would be false).
In that case, the next character must be the start of an identifier
or a "X" sign in %NAME. So in order to
interpret "\" as current value, we check here that the next token
is not the start of an identifier or a "X". (Note - in C, "X" is
an operator, so for example "\X3" means "current location mod 3",
so C is special-cased below).

"'" is an operator in some languages. The only one where it is
a prefix operator (the case that causes ambiguities)
is PLI, where it means "not". We resolve this by simply


```

5208
5209
5210
5211
5212
5213
5214
5215
5216
5217
5218
5219
5220
5221
5222
5223
5224
5225
5226
5227
5228
5229
5230
5231
5232
5233
5234
5235
5236
5237
5238
5239
5240
5241
5242
5243
5244
5245
5246
5247
5248
5249
5250
5251
5252
5253
5254
5255
5256
5257
5258
5259
5260
5261
5262
5263
5264

```

```

! assuming that "" means prevloc if this is an address expression,
! and means "not" in a language expression. This essentially
! disallows "" for prevloc in language expressions (%PREVLOC can
! be used instead).

```

```

"" is highly overloaded. In many languages it can be the start
of a floating point number, so we check for the next character
being a digit here. It can also be the indirection operator
in an address expression. We check for the next character being
"(" or "\", and if so, assume that the dot means
indirection and not current location.

```

```

IF (NOT .CHARTBL[.CHARPTR[0], CHRTBL$V_IDENT_START]) AND
(NOT .CHARTBL[.CHARPTR[0], CHRTBL$V_DIGIT]) AND
(.CHARPTR[0] NEQ '%' OR
(.DBG$GB_LANGUAGE EQL DBG$K_C AND (NOT .ADDRESS_EXPRESSION)))

```

THEN

BEGIN

IF .STARTPTR[0] EQL '\' THEN RETURN CURVAL_TOKEN;

IF (.STARTPTR[0] EQL '(') AND

((.DBG\$GB_LANGUAGE NEQ DBG\$K_PLI) OR .ADDRESS_EXPRESSION)

THEN

RETURN PREVLOC_TOKEN;

IF .STARTPTR[0] EQL '.' AND

NOT ((.CHARPTR[0] EQL '(') OR

(.CHARPTR[0] EQL '.') OR

(.CHARPTR[0] EQL '\'))

THEN

RETURN CURLOC_TOKEN;

END;

CHARPTR = .STARTPTR;

END;

```

! If we are expecting an operand next, check for the special DEBUG symbols
! that begin with a percent sign '%'. This includes %LINE, %LABEL, %NAME,
! and all the register names.

```

```

IF (.CHARPTR[0] EQL '%') AND
.OPERAND_EXPECTED AND
.CHARTBL[.CHARPTR[1], CHRTBL$V_ALPHABETIC]

```

THEN

BEGIN

! Accumulate the identifier after the '%'-sign.

CHARPTR = .CHARPTR + 1;

TOKENLEN = 1;

TOKENBUFFER[1] = '%';

WHILE .CHARTBL[.CHARPTR[0], CHRTBL\$V_ALPHABETIC] OR

.CHARTBL[.CHARPTR[0], CHRTBL\$V_DIGIT]

DO

BEGIN

TOKENLEN = .TOKENLEN + 1;


```

5265
5266
5267
5268
5269
5270
5271
5272
5273
5274
5275
5276
5277
5278
5279
5280
5281
5282
5283
5284
5285
5286
5287
5288
5289
5290
5291
5292
5293
5294
5295
5296
5297
5298
5299
5300
5301
5302
5303
5304
5305
5306
5307
5308
5309
5310
5311
5312
5313
5314
5315
5316
5317
5318
5319
5320
5321

```

```

    TOKENBUFFER[.TOKENLEN] = .CHARPTR[0];
    CHARPTR = .CHARPTR + 1;
    END;

    TOKENBUFFER[0] = .TOKENLEN;

    ! Now look up the '%' symbol in the DEBUG Percent Table.
    !
    INDEX = PERCENT_NOFIND;
    INCR I FROM 0 TO .PERCENT_TABLE[-1] - 1 DO
    BEGIN
        NAMEPTR = .PERCENT_TABLE[I] + TABLEBASE;
        IF CHSEQ(.NAMEPTR[1], NAMEPTR[2], .TOKENLEN, TOKENBUFFER[1], 0)
        THEN
            BEGIN
                INDEX = .NAMEPTR[0];
                EXITLOOP;
            END;
    END;

    ! Now do whatever further processing is appropriate for this '%' symbol.
    !
    CASE .INDEX FROM PERCENT_NOFIND TO PERCENT_IDENT OF
    SET

    ! Handle the No-Find case. We do not recognize this '%' symbol,
    ! so we do nothing.
    [PERCENT_NOFIND]:
        0;

    ! Handle %LINE and %LABEL. Here we pick up the line or label
    ! number that follows the keyword and construct an Identifier
    ! Token Entry for the line or label and return that to the caller.
    [PERCENT_LINE,
    PERCENT_LABEL]:
        BEGIN

        ! Set up the fully spelled out keyword (%LINE or %LABEL) in
        ! TOKENBUFFER and set up the appropriate error message code.
        !
        IF .INDEX EQL PERCENT_LINE
        THEN
            BEGIN
                CH$MOVE(6, UPLIT BYTE(%ASCII '%LINE '), TOKENBUFFER[1]);
                TOKENLEN = 6;
                ERRORMSG = DBGS_SYNERRLINE;
            END;
        ELSE

```

```
BEGIN
CHSMOVE(7, UPLIT BYTE(ASCII '%LABEL '), TOKENBUFFER[1]);
TOKENLEN = 7;
ERRORMSG = DBGS_SYNERRLABEL;
END;

! Check for and scan past the blanks after the %LINE or %LABEL
! keyword and before the line or label number.
IF NOT .CHARTBL[.CHARPTR[0], CHRTBL$V_SPACE] THEN SIGNAL(.ERRORMSG);
WHILE .CHARTBL[.CHARPTR[0], CHRTBL$V_SPACE] DO
    CHARPTR = .CHARPTR + 1;

! Check that a digit follows. Then strip off leading zeroes.
IF NOT .CHARTBL[.CHARPTR[0], CHRTBL$V_DIGIT]
THEN
    SIGNAL(.ERRORMSG);
WHILE (.CHARPTR[0] EQL '0') AND
    .CHARTBL[.CHARPTR[1], CHRTBL$V_DIGIT]
DO
    CHARPTR = .CHARPTR + 1;

! Now pick up the line or label number and copy all the digits
! into TOKENBUFFER.
WHILE .CHARTBL[.CHARPTR[0], CHRTBL$V_DIGIT] DO
    BEGIN
        IF .TOKENLEN GEQ 255 THEN SIGNAL(.ERRORMSG);
        TOKENLEN = .TOKENLEN + 1;
        TOKENBUFFER[.TOKENLEN] = .CHARPTR[0];
        CHARPTR = .CHARPTR + 1;
    END;

! If we are picking up a line number, we also allow a dot
! followed by a statement number (e.g., %LINE 10.2). Pick up
! the statement number if present.
IF (.INDEX EQL PERCENT_LINE) AND (.CHARPTR[0] EQL '.')
THEN
    BEGIN
        ! Check for valid syntax and pick up the dot.
        IF (NOT .CHARTBL[.CHARPTR[1], CHRTBL$V_DIGIT]) OR
            (.TOKENLEN GEQ 255)
        THEN
            SIGNAL(.ERRORMSG);
        TOKENLEN = .TOKENLEN + 1;
        TOKENBUFFER[.TOKENLEN] = '.';
```

```
CHARPTR = .CHARPTR + 1;
```

```
! Strip off leading zeroes.
```

```
WHILE (.CHARPTR[0] EQL '0') AND  
      .CHARTBL[.CHARPTR[1], CHRTBL$V_DIGIT]  
DO  
  CHARPTR = .CHARPTR + 1;
```

```
! Pick up the statement number itself.
```

```
WHILE .CHARTBL[.CHARPTR[0], CHRTBL$V_DIGIT] DO  
  BEGIN  
    IF .TOKENLEN GEQ 255 THEN SIGNAL(.ERRORMSG);  
    TOKENLEN = .TOKENLEN + 1;  
    TOKENBUFFER[.TOKENLEN] = .CHARPTR[0];  
    CHARPTR = .CHARPTR + 1;  
  END;
```

```
END;
```

```
! We now have a complete %LINE or %LABEL name string. Create  
! and return an Identifier Token Entry for it.
```

```
TOKENBUFFER[0] = .TOKENLEN;  
RETURN CREATE_OPERAND_TOKEN(TOKEN$K_IDENTIFIER, TOKENBUFFER);
```

```
END; ! End of %LINE and %LABEL processing
```

```
! Handle %NAME. Here we pick up the identifier symbol that follows  
! (directly or in quotes) and return an Identifier Token for it.
```

```
[PERCENT NAME]:  
BEGIN
```

```
! Scan past any blanks after the %NAME keyword and before the  
! actual name string itself. Then set STARTPTR to that place.
```

```
WHILE .CHARTBL[.CHARPTR[0], CHRTBL$V_SPACE] DO  
  CHARPTR = .CHARPTR + 1;
```

```
STARTPTR = .CHARPTR;
```

```
! If the name string starts with a quote character, pick up all  
! characters in the quoted string and move them to TOKENBUFFER.
```

```
IF .CHARTBL[.CHARPTR[0], CHRTBL$V_STRING_QUOTE]  
THEN  
  SCAN_QUOTED_STRING(TOKENBUFFER, TOKEN_TYPE)
```

```
! Otherwise, pick up any consecutive string of characters which  
! are allowed anywhere in identifiers. For most languages,  
! this is any string composed of A - Z, 0 - 9, $, and _.
```

```
ELSE
```

```
  BEGIN
```

```
    WHILE .CHARTBL[.CHARPTR[0], CHRTBL$V_IDENT_START] OR  
          .CHARTBL[.CHARPTR[0], CHRTBL$V_IDENT_MIDDLE] OR  
          .CHARTBL[.CHARPTR[0], CHRTBL$V_IDENT_END]
```

```
    DO
```

```
      CHARPTR = .CHARPTR + 1;
```

```
    TOKENLEN = .CHARPTR - .STARTPTR;
```

```
    IF .TOKENLEN EQL 0 THEN SIGNAL(DBG$NAMSTRMIS);
```

```
    CH$MOVE(.TOKENLEN, .STARTPTR, TOKENBUFFER[1]);
```

```
    TOKENBUFFER[0] = .TOKENLEN;
```

```
  END;
```

```
! Create and return an Identifier Token Entry for the symbol.
```

```
RETURN CREATE_OPERAND_TOKEN(TOKEN$K_IDENTIFIER, TOKENBUFFER);
```

```
END;
```

```
! End of %NAME processing
```

```
! Handle '%' symbol names that we recognize as identifiers. This  
! includes all the register names. Return an Identifier Token.
```

```
[PERCENT IDENT]:
```

```
  RETURN CREATE_OPERAND_TOKEN(TOKEN$K_IDENTIFIER, TOKENBUFFER);
```

```
! Handle the %DEC operator. This sets the radix to decimal for  
! the expression which constitutes its operand.
```

```
[PERCENT DEC]:
```

```
  RETURN RADIX_OP_DEC;
```

```
! Handle the %HEX operator. This sets the radix to hexadecimal for  
! the expression which constitutes its operand.
```

```
[PERCENT HEX]:
```

```
  RETURN RADIX_OP_HEX;
```

```
! Handle the %OCT operator. This sets the radix to octal for  
! the expression which constitutes its operand.
```

```
[PERCENT OCT]:
```

```
  RETURN RADIX_OP_OCT;
```

```
! Handle the %BIN operator. This sets the radix to binary for  
! the expression which constitutes its operand.
```



```
5493 5604 [PERCENT BIN]:
5494 5605 RETURN RADIX_OP_BIN;
5495 5606
5496 5607
5497 5608 ! Any other CASE index is an internal DEBUG error.
5498 5609
5499 5610 [INRANGE, OUTFRANGE]:
5500 5611 $DBG_ERROR('DBGPARSER\LEXICAL_SCANNER 20');
5501 5612
5502 5613 TES;
5503 5614
5504 5615
5505 5616 ! This is the end of the '%' symbol processing. If we have not recog-
5506 5617 nized the symbol yet, it is not a DEBUG special symbol so we reset
5507 5618 CHARPTR to point to the '%' sign again. The hope is that we will
5508 5619 recognize it as a valid token later in the Lexical Scanner.
5509 5620
5510 5621 CHARPTR = .STARTPTR;
5511 5622
5512 5623 END; ! End of '%' symbol scanning
5513 5624
5514 5625
5515 5626 ! See if this token is a quoted character string. Quoted character strings
5516 5627 in this context are defined to start with a quote character (usually '"' or
5517 5628 ') to continue as a string of zero or more non-quote characters, and to
5518 5629 be terminated by a quote character. The starting and ending quote char-
5519 5630 acters must be the same character and that quote character is represented
5520 5631 within the string by two consecutive quote characters. If we find such
5521 5632 a quoted string, we accumulate it and return a String Constant Token.
5522 5633
5523 5634 IF .CHARTBL[CHARPTR[0], CHRTBL$V_STRING_QUOTE]
5524 5635 THEN
5525 5636 BEGIN
5526 5637 SCAN_QUOTED_STRING(TOKENBUFFER, TOKEN_TYPE);
5527 5638 RETURN CREATE_OPERAND_TOKEN(.TOKEN_TYPE, TOKENBUFFER);
5528 5639 END;
5529 5640
5530 5641
5531 5642 ! See if this token is a numeric constant. If so, we pick up the character
5532 5643 representation of the number and return it as a Numeric Constant Lexical
5533 5644 Token Entry. This scan must be done before the Identifier scan below so
5534 5645 that COBOL integers get interpreted as numbers, not identifiers.
5535 5646
5536 5647 This code simulates a Finite-State Machine (FSM) which accepts any valid
5537 5648 numeric constant in the current language. The machine is defined by a
5538 5649 state table where each state has a set of allowed transitions to other
5539 5650 states. Each transition is selected by the next input character and has
5540 5651 an associated action routine defined below. When a numeric constant has
5541 5652 been recognized, a transition is taken whose action routine builds and
5542 5653 returns a Lexical Token Entry for the numeric constant.
5543 5654
5544 5655 IF .CHARTBL[CHARPTR[0], CHRTBL$V_NUMBER_START] AND
5545 5656 ((NOT .ADDRESS_EXPRESSION) OR (.CHARPTR[0] NEQ '.'))
5546 5657 THEN
5547 5658 BEGIN
5548 5659
5549 5660
```

```
5550 5661 ! Start at the Start State for the Finite-State Machine that will scan
5551 5662 ! the number according to the rules of the current language. This
5552 5663 ! start state is ordinarily zero. If the language is ADA and the radix
5553 5664 ! is not decimal, however, we start at the B_START_STATE, which is
5554 5665 ! the entry to the part of the ADA number scanner that picks up
5555 5666 ! based numbers. Also, set up the initial number kind based on the
5556 5667 ! current radix setting assuming it will be an integer.
5557 5668
5558 5669 STATE_INDEX = 0;
5559 5670 IF (.DBG$GB_LANGUAGE EQL DBG$K_ADA) AND
5560 5671 (.EXPRESSION_RADIX NEQ DBG$K_DECIMAL)
5561 5672 THEN
5562 5673     STATE_INDEX = REMEMBER_ADA_B_START_STATE;
5563 5674
5564 5675 NUMBER_KIND = TOKEN$K_INTEGER;
5565 5676 IF .EXPRESSION_RADIX EQL DBG$K_HEX
5566 5677 THEN
5567 5678     NUMBER_KIND = TOKEN$K_HEX_INTEGER;
5568 5679
5569 5680 IF .EXPRESSION_RADIX EQL DBG$K_OCTAL
5570 5681 THEN
5571 5682     NUMBER_KIND = TOKEN$K_OCT_INTEGER;
5572 5683
5573 5684 IF .EXPRESSION_RADIX EQL DBG$K_BINARY
5574 5685 THEN
5575 5686     NUMBER_KIND = TOKEN$K_BIN_INTEGER;
5576 5687
5577 5688 BACKUP_NUMBER_KIND = .NUMBER_KIND;
5578 5689
5579 5690
5580 5691 ! Then loop through the machine states, selecting each next state based
5581 5692 ! on the next input character and performing the appropriate action for
5582 5693 ! each transition, until the whole number is picked up.
5583 5694
5584 5695 WHILE TRUE DO
5585 5696     BEGIN
5586 5697
5587 5698         ! Get the character class code for the next input character.
5588 5699         ! Small kludge for ADA: In X[1..5], we want to treat '..'
5589 5700         ! as terminating the '1', not as part of the number.
5590 5701
5591 5702         CLASS = .CHARTBL[.CHARPTR[0], CHRTBL$V_NUMBER_CLASS];
5592 5703         IF .DBG$GB_LANGUAGE EQL DBG$K_ADA
5593 5704         THEN
5594 5705             IF .CHARPTR[0] EQL '.' AND .CHARPTR[1] EQL '.'
5595 5706             THEN
5596 5707                 CLASS = NUMST$K_CLASS_OTHER;
5597 5708
5598 5709
5599 5710
5600 5711         ! Loop through the transitions from this state until we find a
5601 5712         ! transition for this character class or for NUMST$K_CLASS_OTHER
5602 5713         ! (the class of all other characters). Pick up the action index
5603 5714         ! and next state associated with this transition of the Finite-
5604 5715         ! State Machine.
5605 5716
5606 5717     WHILE (.STATE_TABLE[.STATE_INDEX, NUMST$B_CHAR_CLASS] NEQ
```

```
.. 5607      5718 4
.. 5608      5719 5
.. 5609      5720 4
.. 5610      5721 4
.. 5611      5722 4
.. 5612      5723 4
.. 5613      5724 4
.. 5614      5725 4
.. 5615      5726 4
.. 5616      5727 4
.. 5617      5728 4
.. 5618      5729 4
.. 5619      5730 4
.. 5620      5731 4
.. 5621      5732 4
.. 5622      5733 4
.. 5623      5734 4
.. 5624      5735 4
.. 5625      5736 4
.. 5626      5737 4
.. 5627      5738 4
.. 5628      5739 4
.. 5629      5740 4
.. 5630      5741 4
.. 5631      5742 4
.. 5632      5743 4
.. 5633      5744 4
.. 5634      5745 4
.. 5635      5746 4
.. 5636      5747 4
.. 5637      5748 4
.. 5638      5749 4
.. 5639      5750 4
.. 5640      5751 4
.. 5641      5752 4
.. 5642      5753 4
.. 5643      5754 4
.. 5644      5755 4
.. 5645      5756 4
.. 5646      5757 4
.. 5647      5758 4
.. 5648      5759 4
.. 5649      5760 5
.. 5650      5761 5
.. 5651      5762 5
.. 5652      5763 4
.. 5653      5764 4
.. 5654      5765 4
.. 5655      5766 4
.. 5656      5767 4
.. 5657      5768 4
.. 5658      5769 4
.. 5659      5770 4
.. 5660      5771 4
.. 5661      5772 4
.. 5662      5773 4
.. 5663      5774 4

      NUMST$K (CLASS OTHER) AND
      (.STATE_TABLE[.STATE_INDEX, NUMST$B_CHAR_CLASS] NEQ .CLASS)
DO
  STATE_INDEX = .STATE_INDEX + 1;
ACTION = .STATE_TABLE[.STATE_INDEX, NUMST$B_ACTION];
STATE_INDEX = .STATE_TABLE[.STATE_INDEX, NUMST$W_NEXT_STATE];

! Case on the action code to select the appropriate semantic
! action routine for this state transition.
CASE .ACTION FROM NUMST$K_MIN_ACTION TO NUMST$K_MAX_ACTION OF
SET

  ! Some state transitions require no semantic action, so we
  ! provide a do-nothing action routine.
[NUMST$K_ACT_DO_NOTHING]:
  0;

  ! Go past a digit in the integer part of the number. Set the
  ! digit backup pointer to the last good digit seen (this one).
[NUMST$K_ACT_GO_PAST_DIGIT]:
  BACKUP_DIGIT_PTR = .CHARPTR;

  ! At the decimal point, mark that we have a floating-point
  ! number.
[NUMST$K_ACT_MARK_DEC_PT]:
  NUMBER_KIND = TOKEN$K_FLOATING;

  ! At a fraction digit, set the backup pointer to the last good
  ! digit seen (i.e., this one) and note that we really have a
  ! floating-point number even if we must back up the scan.
[NUMST$K_ACT_GO_PAST_FRAC]:
  BEGIN
  BACKUP_DIGIT_PTR = .CHARPTR;
  BACKUP_NUMBER_KIND = TOKEN$K_FLOATING;
  END;

  ! Mark that we found an E exponent marker.
[NUMST$K_ACT_MARK_E_EXP]:
  NUMBER_KIND = TOKEN$K_EXP_E_FLOAT;

  ! Mark that we found a D exponent marker.
[NUMST$K_ACT_MARK_D_EXP]:
```



```
.. 5664 5775 4
.. 5665 5776 4
.. 5666 5777 4
.. 5667 5778 4
.. 5668 5779 4
.. 5669 5780 4
.. 5670 5781 4
.. 5671 5782 4
.. 5672 5783 4
.. 5673 5784 4
.. 5674 5785 4
.. 5675 5786 4
.. 5676 5787 4
.. 5677 5788 4
.. 5678 5789 4
.. 5679 5790 4
.. 5680 5791 4
.. 5681 5792 4
.. 5682 5793 4
.. 5683 5794 4
.. 5684 5795 5
.. 5685 5796 5
.. 5686 5797 6
.. 5687 5798 6
.. 5688 5799 6
.. 5689 5800 5
.. 5690 5801 5
.. 5691 5802 4
.. 5692 5803 4
.. 5693 5804 4
.. 5694 5805 4
.. 5695 5806 4
.. 5696 5807 4
.. 5697 5808 4
.. 5698 5809 4
.. 5699 5810 5
.. 5700 5811 5
.. 5701 5812 6
.. 5702 5813 6
.. 5703 5814 6
.. 5704 5815 5
.. 5705 5816 5
.. 5706 5817 5
.. 5707 5818 5
.. 5708 5819 5
.. 5709 5820 4
.. 5710 5821 4
.. 5711 5822 4
.. 5712 5823 4
.. 5713 5824 4
.. 5714 5825 4
.. 5715 5826 4
.. 5716 5827 4
.. 5717 5828 4
.. 5718 5829 4
.. 5719 5830 5
.. 5720 5831 5
```

```
NUMBER_KIND = TOKENSK_EXP_D_FLOAT;
```

```
! Mark that we found a G exponent marker.
```

```
[NUMSTSK ACT MARK_G_EXP]:
```

```
NUMBER_KIND = TOKENSK_EXP_G_FLOAT;
```

```
! Mark that we found a Q exponent marker.
```

```
[NUMSTSK ACT MARK_Q_EXP]:
```

```
NUMBER_KIND = TOKENSK_EXP_Q_FLOAT;
```

```
! At a pack decimal digit, set the backup pointer to the last good
digit seen (i.e., this one) and note that we really have a
pack decimal number even if we must back up the scan.
```

```
[NUMSTSK ACT_GO_PAST_PACK]:
```

```
BEGIN
```

```
BACKUP_DIGIT_PTR = .CHARPTR;
```

```
IF NOT (.NUMBER_KIND EQL TOKENSK_HEX_INTEGER OR
        .NUMBER_KIND EQL TOKENSK_OCT_INTEGER OR
        .NUMBER_KIND EQL TOKENSK_BIN_INTEGER)
```

```
THEN
```

```
BACKUP_NUMBER_KIND = TOKENSK_PACK_DECIMAL;
```

```
END;
```

```
! At a pack decimal digit, set the backup pointer to the last good
digit seen (i.e., this one) and note that we really have a
pack decimal number even if we must back up the scan.
```

```
[NUMSTSK ACT_GO_PAST_PACK_FRAC]:
```

```
BEGIN
```

```
BACKUP_DIGIT_PTR = .CHARPTR;
```

```
IF NOT (.NUMBER_KIND EQL TOKENSK_HEX_INTEGER OR
        .NUMBER_KIND EQL TOKENSK_OCT_INTEGER OR
        .NUMBER_KIND EQL TOKENSK_BIN_INTEGER)
```

```
THEN
```

```
BACKUP_NUMBER_KIND = TOKENSK_PACK_DECIMAL
```

```
ELSE
```

```
BACKUP_NUMBER_KIND = TOKENSK_FLOATING;
```

```
END;
```

```
! We have a number but we scanned too far. This can happen in
FORTRAN with "25.EQ." where we pick up the "." and "E" since
that could be part of a number. Back up the scan pointer to
the true end of the number and return the appropriate Numeric
Constant Lexical Token Entry.
```

```
[NUMSTSK ACT_BACKUP_PTRS]:
```

```
BEGIN
```

```
NUMBER_KIND = .BACKUP_NUMBER_KIND;
```



```
CHARPTR = .BACKUP_DIGIT_PTR + 1;
TOKENLEN = .CHARPTR - .STARTPTR;
IF .TOKENLEN GTR 255 THEN SIGNAL(DBG$NUMCONLONG);
TOKENBUFFER[0] = .TOKENLEN;
CH$MOVE(.TOKENLEN, .STARTPTR, TOKENBUFFER[1]);
RETURN CREATE_OPERAND_TOKEN(.NUMBER_KIND, TOKENBUFFER);
END;
```

```
! We have a complete and valid numeric constant. Create a
! Numeric Constant Lexical Token Entry for it and return a
! pointer to that Token Entry to the caller.
```

```
[NUM$K_ACT_GOT_NUMBER]:
BEGIN
  TOKENLEN = .CHARPTR - .STARTPTR;
  IF .TOKENLEN GTR 255 THEN SIGNAL(DBG$NUMCONLONG);
  TOKENBUFFER[0] = .TOKENLEN;
  CH$MOVE(.TOKENLEN, .STARTPTR, TOKENBUFFER[1]);
  RETURN CREATE_OPERAND_TOKEN(.NUMBER_KIND, TOKENBUFFER);
END;
```

```
! We have a complete and valid pack decimal constant. Create a
! Numeric Constant Lexical Token Entry for it and return a
! pointer to that Token Entry to the caller.
```

```
[NUM$K_ACT_GOT_PACK_NUMBER]:
BEGIN
  IF NOT (.NUMBER_KIND EQL TOKEN$K_HEX_INTEGER OR
          .NUMBER_KIND EQL TOKEN$K_OCT_INTEGER OR
          .NUMBER_KIND EQL TOKEN$K_BIN_INTEGER)
  THEN
    NUMBER_KIND = TOKEN$K_PACK_DECIMAL;

  TOKENLEN = .CHARPTR - .STARTPTR;
  IF .TOKENLEN GTR 255 THEN SIGNAL(DBG$NUMCONLONG);
  TOKENBUFFER[0] = .TOKENLEN;
  CH$MOVE(.TOKENLEN, .STARTPTR, TOKENBUFFER[1]);
  RETURN CREATE_OPERAND_TOKEN(.NUMBER_KIND, TOKENBUFFER);
END;
```

```
! This is not a valid numeric constant. Back up CHARPTR to the
! start of the lexical token and exit the Number Scanner code.
```

```
[NUM$K_ACT_NOT_NUMBER]:
BEGIN
  CHARPTR = .STARTPTR;
  EXITLOOP;
END;
```

```
! In cobol, 12A will be a name if the mode setting is not hex.
```

```
[NUM$K_ACT_COB_CKHEX]:
BEGIN
```

```
5721 5832 5
5722 5833 5
5723 5834 5
5724 5835 5
5725 5836 5
5726 5837 5
5727 5838 4
5728 5839 4
5729 5840 4
5730 5841 4
5731 5842 4
5732 5843 4
5733 5844 4
5734 5845 4
5735 5846 5
5736 5847 5
5737 5848 5
5738 5849 5
5739 5850 5
5740 5851 5
5741 5852 4
5742 5853 4
5743 5854 4
5744 5855 4
5745 5856 4
5746 5857 4
5747 5858 4
5748 5859 4
5749 5860 5
5750 5861 6
5751 5862 6
5752 5863 6
5753 5864 5
5754 5865 5
5755 5866 5
5756 5867 5
5757 5868 5
5758 5869 5
5759 5870 5
5760 5871 5
5761 5872 4
5762 5873 4
5763 5874 4
5764 5875 4
5765 5876 4
5766 5877 4
5767 5878 4
5768 5879 5
5769 5880 5
5770 5881 5
5771 5882 4
5772 5883 4
5773 5884 4
5774 5885 4
5775 5886 4
5776 5887 4
5777 5888 5
```

```
IF .EXPRESSION_RADIX NEQ DBG$K_HEX
THEN
  BEGIN
    CHARPTR = .STARTPTR;
    EXITLOOP;
  END;
```

```
END;
```

```
! In COBOL, a number such as 123 can be a name. Such a name
! must be entered as $NAME 123. However, if we find a number
! followed by a valid identifier character, as in 123A or 12-3,
! then we actually have an identifier, so we exit the number
! scanning loop without returning a number token. If the
! number ended with any other character, we return a valid
! number token.
```

```
[NUMST$K_ACT_COB_KNUM]:
```

```
  BEGIN
    IF .CHARTBL[.CHARPTR[0], CHRTBL$V_IDENT_START] OR
        .CHARTBL[.CHARPTR[0], CHRTBL$V_IDENT_MIDDLE] OR
        .CHARTBL[.CHARPTR[0], CHRTBL$V_IDENT_END]
    THEN
      BEGIN
        CHARPTR = .STARTPTR;
        EXITLOOP;
      END;
```

```
  IF NOT ((.NUMBER_KIND EQL TOKEN$K_HEX_INTEGER) OR
            (.NUMBER_KIND EQL TOKEN$K_OCT_INTEGER) OR
            (.NUMBER_KIND EQL TOKEN$K_BIN_INTEGER))
```

```
  THEN
    NUMBER_KIND = TOKEN$K_PACK_DECIMAL;
```

```
  TOKENLEN = .CHARPTR - .STARTPTR;
  IF .TOKENLEN GTR 255 THEN SIGNAL(DBG$_NUMCONLONG);
  TOKENBUFFER[0] = .TOKENLEN;
  CH$MOVE(.TOKENLEN, .STARTPTR, TOKENBUFFER[1]);
  RETURN CREATE_OPERAND_TOKEN(.NUMBER_KIND, TOKENBUFFER);
END;
```

```
! Save the base of a number in ADA, where a number can be
! of the form base#number.
```

```
[NUMST$K_ACT_SAVE_BASE]:
```

```
  0; T<<<-----
```

```
! Any other action index constitutes an internal error.
```

```
[NUMST$K_ACT_GIVE_ERROR,
```

```
  INRANGE
  OUTRANGE]:
  $DBG_ERROR('DBGPARSER\LEXICAL_SCANNER 30');
```

```
5778 5889 5
5779 5890 5
5780 5891 6
5781 5892 6
5782 5893 6
5783 5894 5
5784 5895 5
5785 5896 4
5786 5897 4
5787 5898 4
5788 5899 4
5789 5900 4
5790 5901 4
5791 5902 4
5792 5903 4
5793 5904 4
5794 5905 4
5795 5906 4
5796 5907 4
5797 5908 5
5798 5909 5
5799 5910 5
5800 5911 5
5801 5912 5
5802 5913 6
5803 5914 6
5804 5915 6
5805 5916 5
5806 5917 5
5807 5918 6
5808 5919 6
5809 5920 6
5810 5921 5
5811 5922 5
5812 5923 5
5813 5924 5
5814 5925 5
5815 5926 5
5816 5927 5
5817 5928 5
5818 5929 4
5819 5930 4
5820 5931 4
5821 5932 4
5822 5933 4
5823 5934 4
5824 5935 4
5825 5936 4
5826 5937 4
5827 5938 4
5828 5939 4
5829 5940 4
5830 5941 4
5831 5942 4
5832 5943 4
5833 5944 4
5834 5945 4
```

TES;

! Go on to the next character in the buffer and loop.

CHARPTR = .CHARPTR + 1;

END; ! End of WHILE loop over number states

END; ! End of numeric constant scanning

! See if this token is an identifier according to the rules of the current language. If so, pick up the whole identifier, see if this identifier is actually an operator (such as AND or MOD) in the current language, and return either an Identifier Token or an Operator Token.

IF .CHARTBL[CHARPTR], CHRTBL\$V_IDENT_START]

THEN

BEGIN

! We have a valid start character for an identifier in the current language. Now scan through the identifier as long as we have valid middle characters and set ENDPTR each time we find a valid end character for an identifier. At the end of the scan we set CHARPTR to point to the first character after the identifier end character.

ENDPTR = .CHARPTR - 1;

WHILE TRUE DO

BEGIN

IF .CHARTBL[CHARPTR], CHRTBL\$V_IDENT_END]

THEN

BEGIN

ENDPTR = .CHARPTR;

IF NOT .CHARTBL[CHARPTR], CHRTBL\$V_IDENT_MIDDLE] THEN EXITLOOP;

END;

CHARPTR = .CHARPTR + 1;

IF (NOT .CHARTBL[CHARPTR], CHRTBL\$V_IDENT_MIDDLE]) AND

(NOT .CHARTBL[CHARPTR], CHRTBL\$V_IDENT_END])

THEN EXITLOOP;

END;

CHARPTR = .ENDPTR + 1;

! Copy the identifier name to TOKENBUFFER.

TOKENLEN = .CHARPTR - .STARTPTR;

IF .TOKENLEN GTR 255 THEN SIGNAL(DBG\$ IDENTLONG);

CH\$MOVE(.TOKENLEN, .STARTPTR, TOKENBUFFER[1]);

TOKENBUFFER[0] = .TOKENLEN;

! If the language is C, then we want to make sure that the identifier preserves the original casing (upper/lower). This is because

```

5892
5893
5894
5895
5896
5897
5898
5899
5900
5901
5902
5903
5904
5905
5906
5907
5908
5909
5910
5911
5912
5913
5914
5915
5916
5917
5918
5919
5920
5921
5922
5923
5924
5925
5926
5927
5928
5929
5930
5931
5932
5933
5934
5935
5936
5937
5938
5939
5940
5941
5942
5943
5944
5945
5946
5947
5948

```

```

6003
6004
6005
6006
6007
6008
6009
6010
6011
6012
6013
6014
6015
6016
6017
6018
6019
6020
6021
6022
6023
6024
6025
6026
6027
6028
6029
6030
6031
6032
6033
6034
6035
6036
6037
6038
6039
6040
6041
6042
6043
6044
6045
6046
6047
6048
6049
6050
6051
6052
6053
6054
6055
6056
6057
6058
6059

```

```

! in C, upper case XXX is a different variable from lower case xxx,
! for example. So we copy characters from the original command
! buffer instead of the up-cased command buffer.
! We use a flag "CASING_SIGNIFICANT" which is set in the SET_LANGUAGE
! routine for this. At present, C is the only language that sets
! this flag to TRUE.
IF .CASING_SIGNIFICANT
THEN
  BEGIN
    IF (.STARTPTR LSS .DBG$GL_UPCASE_COMMAND_PTR[0]) OR
      (.STARTPTR GTR .DBG$GL_UPCASE_COMMAND_PTR[1])
    THEN
      $DBG_ERROR('DBGPARSER\DBG$LEXICAL_SCANNER 40');

      NEW_STARTPTR = (.STARTPTR - .DBG$GL_UPCASE_COMMAND_PTR[0]) +
        .DBG$GL_ORIG_COMMAND_PTR;
      CH$MOVE(.TOKENLEN, .NEW_STARTPTR, TOKENBUFFER[1]);
    END;

! See if this identifier is actually the name of an operator. We scan
! a language-specific operator table to determine this. If so, return
! the corresponding Operator Token.
BEST_TOKEN_FOUND = 0;
INCR=1 FROM 0 TO .IDENT_OPERATOR_TABLE[-1] - 1 DO
  BEGIN
    TOKEN = .IDENT_OPERATOR_TABLE[1] + TABLEBASE;
    IF CH$EQL(.TOKEN[TOKEN$B_OPLEN],
      TOKEN[TOKEN$A_OPNAME], .TOKENLEN, TOKENBUFFER[1], 0)
    THEN
      BEGIN
        IF (.OPERAND_EXPECTED AND
          (.TOKEN[TOKEN$B_KIND] EQL TOKEN$K_PREFIX_OP)) OR
          ((NOT .OPERAND_EXPECTED) AND
            (.TOKEN[TOKEN$B_KIND] NEQ TOKEN$K_PREFIX_OP))
        THEN
          RETURN .TOKEN
        ELSE
          BEST_TOKEN_FOUND = .TOKEN;
        END;
      END;
    END;

  IF .BEST_TOKEN_FOUND NEQ 0 THEN RETURN .BEST_TOKEN_FOUND;

! See if this identifier is the name of a built-in function. We scan
! a language-specific built-in function table to determine this. If
! so, return the corresponding Operand Token.
BEST_TOKEN_FOUND = 0;
INCR=1 FROM 0 TO .BIF_TABLE[-1] - 1 DO
  BEGIN

```


5949 6060
5950 6061
5951 6062
5952 6063
5953 6064
5954 6065
5955 6066
5956 6067
5957 6068
5958 6069
5959 6070
5960 6071
5961 6072
5962 6073
5963 6074
5964 6075
5965 6076
5966 6077
5967 6078
5968 6079
5969 6080
5970 6081
5971 6082
5972 6083
5973 6084
5974 6085
5975 6086
5976 6087
5977 6088
5978 6089
5979 6090
5980 6091
5981 6092
5982 6093
5983 6094
5984 6095
5985 6096
5986 6097
5987 6098
5988 6099
5989 6100
5990 6101
5991 6102
5992 6103
5993 6104
5994 6105
5995 6106
5996 6107
5997 6108
5998 6109
5999 6110
6000 6111
6001 6112
6002 6113
6003 6114
6004 6115
6005 6116

```
TOKEN = .BIF TABLE[.I] + TABLEBASE;
IF CH$EQL(.TOKEN[TOKEN$B_LENGTH], .TOKEN[TOKEN$A_NAME],
        .TOKENLEN, TOKENBUFFER[1], 0)
THEN
    IF .OPERAND_EXPECTED
    THEN
        RETURN .TOKEN
    ELSE
        BEST_TOKEN_FOUND = .TOKEN;
END;
IF .BEST_TOKEN_FOUND NEQ 0 THEN RETURN .BEST_TOKEN_FOUND;

! It is not an operator or a built-in function. Hence we return an
! Identifier Token for the symbol. Note that we do not accept a
! zero-length identifier (one that has no valid end character).
IF .TOKENLEN NEQ 0
THEN
    RETURN CREATE_OPERAND_TOKEN(TOKEN$K_IDENTIFIER, TOKENBUFFER);
END;
! End of identifier scanning

! See if this token is an operator symbol (such as +, *, /, or :=) in the
! current language. If so, return an Operator Token to the caller.
IF .CHARTBL[.CHARPTR[0], CHRTBL$V_OPCHAR]
THEN
    BEGIN

    ! Determine where the operator symbol ends. By classifying the avail-
    ! able operator characters as prefix, infix, or postfix characters, we
    ! can ensure that we break operator character strings apart at postfix-
    ! infix, infix-prefix, and postfix-prefix boundaries. An operator char-
    ! acter which does not fall in any of those classes cannot be combined
    ! with other characters and is always a symbol by itself (such as '(').
    WHILE .CHARTBL[.CHARPTR[0], CHRTBL$V_OPCHAR_PREFIX] DO
        CHARPTR = .CHARPTR + 1;
    WHILE .CHARTBL[.CHARPTR[0], CHRTBL$V_OPCHAR_INFIX] DO
        CHARPTR = .CHARPTR + 1;
    WHILE .CHARTBL[.CHARPTR[0], CHRTBL$V_OPCHAR_POSTFIX] DO
        CHARPTR = .CHARPTR + 1;
    IF .CHARPTR EQLU .STARTPTR THEN CHARPTR = .CHARPTR + 1;

    ! Copy the operator symbol to TOKENBUFFER.
    TOKENLEN = .CHARPTR - .STARTPTR;
    IF .TOKENLEN GTR 255 THEN SIGNAL(DBG$_INVOPSYM);
    TOKENBUFFER[0] = .TOKENLEN;
```

6006 6117
6007 6118
6008 6119
6009 6120
6010 6121
6011 6122
6012 6123
6013 6124
6014 6125
6015 6126
6016 6127
6017 6128
6018 6129
6019 6130
6020 6131
6021 6132
6022 6133
6023 6134
6024 6135
6025 6136
6026 6137
6027 6138
6028 6139
6029 6140
6030 6141
6031 6142
6032 6143
6033 6144
6034 6145
6035 6146
6036 6147
6037 6148
6038 6149
6039 6150
6040 6151
6041 6152
6042 6153
6043 6154
6044 6155
6045 6156
6046 6157
6047 6158
6048 6159
6049 6160
6050 6161
6051 6162
6052 6163
6053 6164
6054 6165
6055 6166
6056 6167
6057 6168
6058 6169
6059 6170
6060 6171
6061 6172
6062 6173

CHSMOVE(.TOKENLEN, .STARTPTR, TOKENBUFFER[1]);

! We now have an operator symbol in TOKENBUFFER. Look it up in the
! operator table for the current language, and if found, return the
! corresponding Operator Token. Note that we only accept an operator
! if it is a Primary operator (which we accept in both language and
! address expressions) or if we are in a language expression.

BEST_TOKEN_FOUND = 0;

INCR I FROM 0 TO .OPCHAR_OPERATOR_TABLE[-1] - 1 DO

BEGIN

TOKEN = .OPCHAR_OPERATOR_TABLE[I] + TABLEBASE;

IF (.TOKEN[TOKEN\$V_PRIMARY] OR (NOT .ADDRESS_EXPRESSION)) AND

CHSEQL(.TOKEN[TOKEN\$B_OPLEN],
TOKEN[TOKEN\$A_OPNAME], .TOKENLEN, TOKENBUFFER[1], 0)

THEN

BEGIN

IF (.OPERAND_EXPECTED AND

(.TOKEN[TOKEN\$B_KIND] EQL TOKEN\$K_PREFIX_OP)) OR

((NOT .OPERAND_EXPECTED) AND

(.TOKEN[TOKEN\$B_KIND] NEQ TOKEN\$K_PREFIX_OP))

THEN

RETURN .TOKEN

ELSE

BEST_TOKEN_FOUND = .TOKEN;

END;

END;

! If we found an operator but it is not in a valid context (e.g. infix
! operator when we expect a prefix operator), we return it anyway but
! only if it cannot be an address expression operator (which might be
! valid in the current context).

IF (.BEST_TOKEN_FOUND NEQ 0) AND

NOT (.ADDRESS_EXPRESSION AND

.CHARTBL[.STARTPTR[0], CHRTBL\$V_ADDRESS_OP])

THEN

RETURN .BEST_TOKEN_FOUND;

! There is no such operator. Reset CHARPTR to point to the start of
! the token to give the code below a chance to make sense of it.

CHARPTR = .STARTPTR;

END;

! End of operator scanning

! If we are parsing an Address Expression at present, we check for the
! specific operators allowed in Address Expressions. We do not use the
! language-specific rules for combining operator characters in this case.
! If we find such an operator here, return the corresponding Token Entry.

```
6063 6174 : If not, reset CHARPTR and continue through the Lexical Scanner.
6064 6175 :
6065 6176 IF .ADDRESS_EXPRESSION AND .CHARTBL[CHARPTR[0], CHRTBL$V_ADDRESS_OP]
6066 6177 THEN
6067 6178 BEGIN
6068 6179 : Special cases for C.
6069 6180 :
6070 6181 IF (.DBG$GB_LANGUAGE EQL DBG$K_C) AND
6071 6182 (.CHARPTR[0] EQL '-') AND (.CHARPTR[1] EQL '>')
6072 6183 THEN
6073 6184 BEGIN
6074 6185 CHARPTR = .CHARPTR + 2;
6075 6186 RETURN C_ARROW_TOKEN;
6076 6187 END;
6077 6188 IF (.DBG$GB_LANGUAGE EQL DBG$K_C) AND
6078 6189 (.CHARPTR[0] EQL '*') AND
6079 6190 (.OPERAND_EXPECTED)
6080 6191 THEN
6081 6192 BEGIN
6082 6193 CHARPTR = .CHARPTR + 1;
6083 6194 RETURN C_INDIRECT_TOKEN;
6084 6195 END;
6085 6196 :
6086 6197 : Check for all other address expression operators.
6087 6198 :
6088 6199 CHARPTR = .CHARPTR + 1;
6089 6200 BEST_TOKEN_FOUND = 0;
6090 6201 INCR I FROM 0 TO .ADDR_EXPR_OPTBL[-1] - 1 DO
6091 6202 BEGIN
6092 6203 TOKEN = .ADDR_EXPR_OPTBL[I] + TABLEBASE;
6093 6204 IF CH$EQL(.TOKEN[TOKEN$B_OPLEN],
6094 6205 TOKEN[TOKEN$A_OPNAME], 1, STARTPTR[0], 0)
6095 6206 THEN
6096 6207 BEGIN
6097 6208 IF (.OPERAND_EXPECTED AND
6098 6209 (.TOKEN[TOKEN$B_KIND] EQL TOKEN$K_PREFIX_OP)) OR
6099 6210 ((NOT .OPERAND_EXPECTED) AND
6100 6211 (.TOKEN[TOKEN$B_KIND] NEQ TOKEN$K_PREFIX_OP))
6101 6212 THEN
6102 6213 RETURN .TOKEN
6103 6214 ELSE
6104 6215 BEST_TOKEN_FOUND = .TOKEN;
6105 6216 END;
6106 6217 END;
6107 6218 IF .BEST_TOKEN_FOUND NEQ 0 THEN RETURN .BEST_TOKEN_FOUND;
6108 6219 CHARPTR = .STARTPTR;
6109 6220 END;
6110 6221 :
6111 6222 : So far we have not determined what kind of token we have here. We thus
6112 6223 : enter some language-specific code in a last-ditch effort to figure out
6113 6224 : what kind of token we have. If the language-specific code can recognize
6114 6225 :
6115 6226 :
6116 6227 :
6117 6228 :
6118 6229 :
6119 6230 :
```

```
6120  
6121  
6122  
6123  
6124  
6125  
6126  
6127  
6128  
6129  
6130  
6131  
6132  
6133  
6134  
6135  
6136  
6137  
6138  
6139  
6140  
6141  
6142  
6143  
6144  
6145  
6146  
6147  
6148  
6149  
6150  
6151  
6152  
6153  
6154  
6155  
6156  
6157  
6158  
6159  
6160  
6161  
6162  
6163  
6164  
6165  
6166  
6167  
6168  
6169  
6170  
6171  
6172  
6173  
6174  
6175  
6176
```

```
6231  
6232  
6233  
6234  
6235  
6236  
6237  
6238  
6239  
6240  
6241  
6242  
6243  
6244  
6245  
6246  
6247  
6248  
6249  
6250  
6251  
6252  
6253  
6254  
6255  
6256  
6257  
6258  
6259  
6260  
6261  
6262  
6263  
6264  
6265  
6266  
6267  
6268  
6269  
6270  
6271  
6272  
6273  
6274  
6275  
6276  
6277  
6278  
6279  
6280  
6281  
6282  
6283  
6284  
6285  
6286  
6287
```

```
! a valid token, it returns a Lexical Token Entry for it.
```

```
CASE .DBG$GB_LANGUAGE FROM DBG$K_MIN_LANGUAGE TO DBG$K_MAX_LANGUAGE OF  
SET
```

```
! Handle Fortran. Here we pick up the operators of the form .XX. or  
! .XXX. (such as .EQ., .NOT., etc.) and return the corresponding  
! Operator Token. These operators are looked up in the FORTRAN  
! Special Operator Table.
```

```
[DBG$K_FORTRAN]:  
BEGIN
```

```
! Check for . as indirection operator in address expressions.
```

```
! IF .ADDRESS_EXPRESSION AND .OPERAND_EXPECTED AND  
! (.CHARPTR[0] EQL '.')  
THEN
```

```
BEGIN  
CHARPTR = .CHARPTR + 1;  
RETURN FORTRAN_INDIRECT_TOKEN;  
END;
```

```
! See if we have one of the FORTRAN relational or boolean  
! operators.
```

```
WHILE .CHARTBL[CHARPTR[1], CHRTBL$V_ALPHABETIC] DO  
CHARPTR = .CHARPTR + 1;
```

```
CHARPTR = .CHARPTR + 2;  
INCR I FROM 0 TO .FORTRAN_SPECIAL_OPTBL[-1] - 1 DO  
BEGIN  
TOKEN = .FORTRAN_SPECIAL_OPTBL[I] + TABLEBASE;  
IF CH$EQL(.TOKEN[TOKEN$B_OPLEN], TOKEN[TOKEN$A_OPNAME],  
CHARPTR - .STARTPTR, .STARTPTR, 0)  
THEN  
RETURN .TOKEN;  
END;
```

```
! Pick up FORTRAN Predefined Constant.
```

```
INCR I FROM 0 TO .PRIDTBL[-1] - 1 DO  
BEGIN  
PRID = .PRIDTBL[I] + TABLEBASE;  
IF CH$EQL(.PRID[PRID$B_LENGTH], PRID[PRID$A_NAME],  
CHARPTR - .STARTPTR, .STARTPTR, 0)  
THEN  
BEGIN  
CH$MOVE(.PRID[PRID$B_LENGTH], PRID[PRID$A_NAME], TOKENBUFFER[1]);  
TOKENBUFFER[0] = .PRID[PRID$B_LENGTH];  
RETURN CREATE_OPERAND_TOKEN(TOKEN$K_IDENTIFIER, TOKENBUFFER);  
END;
```



```

6177
6178
6179
6180
6181
6182
6183
6184
6185
6186
6187
6188
6189
6190
6191
6192
6193
6194
6195
6196
6197
6198
6199
6200
6201
6202
6203
6204
6205
6206
6207
6208
6209
6210
6211
6212
6213
6214
6215
6216
6217
6218
6219
6220
6221
6222
6223
6224
6225
6226
6227
6228
6229
6230
6231
6232
6233

```

```

6288
6289
6290
6291
6292
6293
6294
6295
6296
6297
6298
6299
6300
6301
6302
6303
6304
6305
6306
6307
6308
6309
6310
6311
6312
6313
6314
6315
6316
6317
6318
6319
6320
6321
6322
6323
6324
6325
6326
6327
6328
6329
6330
6331
6332
6333
6334
6335
6336
6337
6338
6339
6340
6341
6342
6343
6344

```

```

END;

! Return "." as the FORTRAN record component selection operator.
! IF (NOT .OPERAND_EXPECTED) AND (.STARTPTR[0] EQL '.')
! THEN
!   BEGIN
!     CHARPTR = .STARTPTR + 1;
!     RETURN FORTRAN_DOT_TOKEN;
!   END;
! END;

! Handle C. Here we pick up the C operators prefix &, infix &,
! infix &&, prefix and postfix ++, prefix and postfix --, +, -, and ->.
! Possible ambiguities involving these operators must be
! resolved using C rules, which is what we do here.
[DBGSK C]:
  BEGIN

    ! Check for the operators that begin with '&': address-of,
    ! bit-and, and short-and.
    ! IF .CHARPTR[0] EQL '&'
    ! THEN
    !   BEGIN
    !     CHARPTR = .CHARPTR + 1;
    !     IF .CHARPTR[0] EQL '&'
    !     THEN
    !       BEGIN
    !         CHARPTR = .CHARPTR + 1;
    !         RETURN C_AND_TOKEN;
    !       END;
    !     IF .OPERAND_EXPECTED THEN RETURN C_ADDR_OF_TOKEN;
    !     RETURN C_BIT_AND_TOKEN;
    !   END;

    ! Check for the operators that start with '+': add, pre-increment,
    ! and post-increment. (X+Y ++X X++)
    ! IF .CHARPTR[0] EQL '+'
    ! THEN
    !   BEGIN
    !     CHARPTR = .CHARPTR + 1;
    !     IF .CHARPTR[0] EQL '+'
    !     THEN
    !       BEGIN
    !         CHARPTR = .CHARPTR + 1;
    !         IF .OPERAND_EXPECTED
    !         THEN

```

```
6234 6345
6235 6346
6236 6347
6237 6348
6238 6349
6239 6350
6240 6351
6241 6352
6242 6353
6243 6354
6244 6355
6245 6356
6246 6357
6247 6358
6248 6359
6249 6360
6250 6361
6251 6362
6252 6363
6253 6364
6254 6365
6255 6366
6256 6367
6257 6368
6258 6369
6259 6370
6260 6371
6261 6372
6262 6373
6263 6374
6264 6375
6265 6376
6266 6377
6267 6378
6268 6379
6269 6380
6270 6381
6271 6382
6272 6383
6273 6384
6274 6385
6275 6386
6276 6387
6277 6388
6278 6389
6279 6390
6280 6391
6281 6392
6282 6393
6283 6394
6284 6395
6285 6396
6286 6397
6287 6398
6288 6399
6289 6400
6290 6401

      | Since ++ is not supported, put out an
      | error message to that effect.
      | If it does become supported then
      | un-comment the commented out code
      | and take out the error signal.
      |
      | SIGNAL(DBG$ SIDEFFECT)
      | ! RETURN C_PRE_INCR_TOKEN
      |
      ELSE
      | SIGNAL(DBG$ SIDEFFECT);
      | ! RETURN C_POST_INCR_TOKEN;
      END;

      RETURN C_ADD_TOKEN;
      END;

      | Check for the operators that start with '-': subtract, pre-decrement,
      | and post-decrement, unary minus, and ->.
      IF .CHARPTR[0] EQL '-'
      THEN
      BEGIN
      CHARPTR = .CHARPTR + 1;
      IF .CHARPTR[0] EQL '-'
      THEN
      BEGIN
      CHARPTR = .CHARPTR + 1;
      IF .OPERAND_EXPECTED
      THEN
      |
      | Since -- is not supported, put out an
      | error message to that effect.
      | If it does become supported then
      | un-comment the commented out code
      | and take out the error signal.
      |
      | SIGNAL(DBG$ SIDEFFECT)
      | ! RETURN C_PRE_DECR_TOKEN
      |
      ELSE
      | SIGNAL(DBG$ SIDEFFECT);
      | ! RETURN C_POST_INCR_TOKEN;
      END;
      IF .CHARPTR[0] EQL '>'
      THEN
      BEGIN
      CHARPTR = .CHARPTR + 1;
      RETURN C_ARROW_TOKEN;
      END;
      IF .OPERAND_EXPECTED
      THEN
```

6291 6402 4
6292 6403 4
6293 6404 4
6294 6405 4
6295 6406 4
6296 6407 4
6297 6408 4
6298 6409 4
6299 6410 4
6300 6411 4
6301 6412 4
6302 6413 4
6303 6414 4
6304 6415 4
6305 6416 4
6306 6417 4
6307 6418 4
6308 6419 4
6309 6420 4
6310 6421 4
6311 6422 4
6312 6423 4
6313 6424 4
6314 6425 4
6315 6426 6
6316 6427 6
6317 6428 6
6318 6429 6
6319 6430 6
6320 6431 6
6321 6432 6
6322 6433 6
6323 6434 6
6324 6435 6
6325 6436 6
6326 6437 7
6327 6438 7
6328 6439 7
6329 6440 8
6330 6441 8
6331 6442 8
6332 6443 7
6333 6444 7
6334 6445 7
6335 6446 7
6336 6447 8
6337 6448 7
6338 6449 6
6339 6450 6
6340 6451 6
6341 6452 6
6342 6453 6
6343 6454 6
6344 6455 6
6345 6456 6
6346 6457 6
6347 6458 6

```
        RETURN C_MINUS_TOKEN
    ELSE
        RETURN C_SUB_TOKEN;
    END;

END;

! Handle RPG. The special indicator names start with '*'.
[DBG$K RPG]:
BEGIN
    IF .CHARPTR[0] EQL '*'
    THEN
        BEGIN
            ! Pick it up as a name.
            IF .OPERAND_EXPECTED
            THEN
                BEGIN
                    CHARPTR = .CHARPTR + 1;
                    IF .CHARTBL[.CHARPTR[0], CHRTBL$V_IDENT_START]
                    THEN
                        BEGIN
                            ! We have a valid start character for an identifier in the current
                            ! language. Now scan through the identifier as long as we have valid
                            ! middle characters and set ENDPTR each time we find a valid end char-
                            ! acter for an identifier. At the end of the scan we set CHARPTR to
                            ! point to the first character after the identifier end character.
                            ENDPTR = .CHARPTR - 2;
                            WHILE TRUE DO
                                BEGIN
                                    IF .CHARTBL[.CHARPTR[0], CHRTBL$V_IDENT_END]
                                    THEN
                                        BEGIN
                                            ENDPTR = .CHARPTR;
                                            IF NOT .CHARTBL[.CHARPTR[0], CHRTBL$V_IDENT_MIDDLE] THEN EXITLOOP;
                                        END;

                                        CHARPTR = .CHARPTR + 1;
                                        IF (NOT .CHARTBL[.CHARPTR[0], CHRTBL$V_IDENT_MIDDLE]) AND
                                            (NOT .CHARTBL[.CHARPTR[0], CHRTBL$V_IDENT_END])
                                        THEN EXITLOOP;
                                    END;

                                CHARPTR = .ENDPTR + 1;

                                ! Copy the identifier name to TOKENBUFFER.
                                !
                                TOKENLEN = .CHARPTR - .STARTPTR;
                                IF .TOKENLEN GTR 255 THEN SIGNAL(DBG$ IDENTLONG);
                                CH$MOVE(.TOKENLEN, .STARTPTR, TOKENBUFFER[1]);
```

```
6348      TOKENBUFFER[0] = .TOKENLEN;  
6349      RETURN CREATE_OPERAND_TOKEN(TOKEN$K_IDENTIFIER, TOKENBUFFER);  
6350      END;  
6351      END  
6352      ! Pick it up as an operator.  
6353      !  
6354      ELSE  
6355      BEGIN  
6356      CHARPTR = .CHARPTR + 1;  
6357      RETURN RPG_MULTIPLY_TOKEN;  
6358      END;  
6359      END;  
6360      END;  
6361      END;  
6362      END;  
6363      END;  
6364      END;  
6365      END;  
6366      ! Handle Ada. Here we distinguish the tick operator from the single  
6367      ! quote--both are represented by the character '"'. If it is a tick,  
6368      ! we return the Tick Operator Token and if it is a single quote, we  
6369      ! pick up the character it quotes and return a Character Constant Token.  
6370      [DBG$K_ADA]:  
6371      BEGIN  
6372      IF .CHARPTR[0] EQL '"'  
6373      THEN  
6374      BEGIN  
6375      ! If we are expecting an infix or postfix operator, this must  
6376      ! be the "tick" character, which begins one of the postfix  
6377      ! tick operators ("FIRST", "LAST", ...).  
6378      IF NOT .OPERAND_EXPECTED  
6379      THEN  
6380      BEGIN  
6381      ! Make a copy of the tick token.  
6382      !  
6383      TOKEN = DBG$GET_TEMPMEM (TOKEN$K_FIXED_SIZE_LONG + 4);  
6384      CH$MOVE (TOKEN$K_FIXED_SIZE_BYTE+1, ADA_TICK_TOKEN, .TOKEN);  
6385      ! Look up which tick operator this is.  
6386      !  
6387      INCR INDEX FROM TOKEN$K_TICK_MIN TO TOKEN$K_TICK_MAX DO  
6388      BEGIN  
6389      NAMEPTR = .ADA_TICK_TABLE[.INDEX] + TABLEBASE;  
6390      IF CH$EQL (.NAMEPTR[0], NAMEPTR[1], .NAMEPTR[0], CHARPTR[1])  
6391      THEN  
6392      BEGIN  
6393      CHARPTR = .CHARPTR + 1 + .NAMEPTR[0];  
6394      TOKEN[TOKEN$K_SUBCODE] = .INDEX;  
6395      TOKEN[TOKEN$K_OPLEN] = .NAMEPTR[0] + 1;  
6396      END;  
6397      END;  
6398      END;  
6399      END;  
6400      END;  
6401      END;  
6402      END;  
6403      END;  
6404      END;
```



```

6405
6406
6407
6408
6409
6410
6411
6412
6413
6414
6415
6416
6417
6418
6419
6420
6421
6422
6423
6424
6425
6426
6427
6428
6429
6430
6431
6432
6433
6434
6435
6436
6437
6438
6439
6440
6441
6442
6443
6444
6445
6446
6447
6448
6449
6450
6451
6452
6453
6454
6455
6456
6457
6458
6459
6460
6461

```

```

6516
6517
6518
6519
6520
6521
6522
6523
6524
6525
6526
6527
6528
6529
6530
6531
6532
6533
6534
6535
6536
6537
6538
6539
6540
6541
6542
6543
6544
6545
6546
6547
6548
6549
6550
6551
6552
6553
6554
6555
6556
6557
6558
6559
6560
6561
6562
6563
6564
6565
6566
6567
6568
6569
6570
6571
6572

```

```

CHSCOPY(1, CHSPTR(UPLIT('')), .NAMEPTR[0], .NAMEPTR[1],
        &C', .NAMEPTR[0]+1, .TOKEN[TOKENSA_OPNAME]);

! Look for a left paren that would indicate a
! list of arguments follow the tick operator.
! If a paren is found, the subcode value is set
! set to the corresponding tick operator type by
! adding 1 to it. Also, the CHARPTR will be
! updated to pointer at the character just past
! the paren.
WHILE .CHARTBL[CHARPTR[0], CHRTBLSV_SPACE] DO
    CHARPTR = .CHARPTR + 1;

IF .CHARPTR[0] EQL '('
THEN
    BEGIN
        TOKEN[TOKENSV_ARGUMENT_LIST] = TRUE;
        CHARPTR = .CHARPTR + 1;
    END
ELSE
    TOKEN[TOKENSV_ARGUMENT_LIST] = FALSE;

RETURN .TOKEN;
END;

END;

! If we fall through to here, we failed to find
! a matching tick operator in our table.
TOKENBUFFER[0] = 1;
TOKENBUFFER[1] = 'i...';
CHARPTR = .CHARPTR + 1;
WHILE .CHARTBL[CHARPTR[0], CHRTBLSV_IDENT_START] OR
        .CHARTBL[CHARPTR[0], CHRTBLSV_IDENT_MIDDLE] OR
        .CHARTBL[CHARPTR[0], CHRTBLSV_IDENT_END] DO
    BEGIN
        IF (.CHARPTR[0] EQL CAR_RET) OR (.TOKENBUFFER[0] GEQ 32)
        THEN EXITLOOP;
        TOKENBUFFER[0] = .TOKENBUFFER[0] + 1;
        TOKENBUFFER[TOKENBUFFER[0]] = .CHARPTR[0];
        CHARPTR = .CHARPTR + 1;
    END;

    SIGNAL(DBG$UNKATTRIB, 1, TOKENBUFFER);
    END;

! Otherwise we are expecting an operand, so it must be the
! single quote character. Pick up the single character quoted
! and return a Character Constant Lexical Token Entry.
IF (.CHARPTR[1] EQL CAR_RET) OR (.CHARPTR[2] NEQ '')
THEN
    SIGNAL(DBG$INVCHRCON);

    TOKENBUFFER[0] = 3;
    CHSMOVE(3, CHARPTR[0], TOKENBUFFER[1]);

```

```
6462      6573      4      CHARPTR = .CHARPTR + 3;
6463      6574      RETURN CREATE_OPERAND_TOKEN(TOKEN$K_IDENTIFIER, TOKENBUFFER);
6464      6575      END;
6465      6576
6466      6577      END;
6467      6578
6468      6579      ! Do nothing for all other languages.
6469      6580
6470      6581      [INRANGE, OUTRANGE]:
6471      6582      0;
6472      6583
6473      6584      TES;
6474      6585
6475      6586
6476      6587
6477      6588      ! We have not found a valid token yet. This must therefore be a genuine
6478      6589      ! syntax error, so we signal an appropriate error message.
6479      6590
6480      6591      CHARPTR = .STARTPTR;
6481      6592      TOKENBUFFER[0] = 0;
6482      6593      INCR I FROM 0 TO 20 DO
6483      6594      BEGIN
6484      6595      IF .CHARPTR[I] EQL CAR RET THEN EXITLOOP;
6485      6596      TOKENBUFFER[I + 1] = .CHARPTR[I];
6486      6597      TOKENBUFFER[0] = .TOKENBUFFER[0] + 1;
6487      6598      END;
6488      6599
6489      6600      SIGNAL(DBG$_SYNERREXPR, 1, TOKENBUFFER);
6490      6601      RETURN 0;
6491      6602
6492      6603      1      END;
INFO#250      L1:5832
: Referenced LOCAL symbol BACKUP_DIGIT_PTR is probably not initialized
```

```
                                .PSECT DBG$PLIT,NOWRT, SHR, PIC,0
49  58  45  4C  5C  52  45  53  52  41  50  47  42  44  1C  03117 P.AWZ: .ASCII <28>\DBGPARSER\<92>\LEXICAL_SCANNER 10\
    30  31  20  52  45  4E  4E  41  43  53  5F  4C  41  43  03126
                                20  4C  45  4E  49  4C  25  03134 P.AXA: .ASCII \XLINE \
49  58  45  4C  5C  52  45  53  20  4C  45  42  41  4C  25  0313A P.AXB: .ASCII \XLABEL \
    30  32  20  52  45  4E  4E  41  43  53  5F  4C  41  43  03141 P.AXC: .ASCII <28>\DBGPARSER\<92>\LEXICAL_SCANNER 20\
49  58  45  4C  5C  52  45  53  52  41  50  47  42  44  1C  03150
    30  33  20  52  45  4E  4E  41  43  53  5F  4C  41  43  0315E P.AXD: .ASCII <28>\DBGPARSER\<92>\LEXICAL_SCANNER 30\
24  47  42  44  5C  52  45  53  52  41  50  47  42  44  20  0316D
52  45  4E  4E  41  43  53  5F  4C  41  43  49  58  45  4C  0317B P.AXE: .ASCII \ DBGPARSER\<92>\DBG$LEXICAL_SCANNER 40\
                                30  34  20  0318A
                                00  00  00  27  0319C P.AXF: .ASCII \'\<0><0><0>
```

```
                                .PSECT DBG$CODE,NOWRT, SHR, PIC,0
                                OFFC 00000
                                .ENTRY DBG$LEXICAL_SCANNER, Save R2,R3,R4,R5,R6,- : 5034
                                R7,R8,R9,R10,R11
```


			1A	11	000EB		BRB	14\$		
	2D		57	91	000ED	13\$:	CMPB	R7, #45	5246	
			15	12	000FO		BNEQ	14\$		
	3E	01	A4	91	000F2		CMPB	1(R4), #62		
			0F	12	000F6		BNEQ	14\$		
00000000'	EF		02	C0	000F8		ADDL2	#2, CHARPTR	5249	
	50	00000000'	EF	9E	000FF		MOVAB	PLI_ARROW_TOKEN, R0	5250	
				04	00106		RET			
0C	AE	04	AC	D0	00107	14\$:	MOVL	OPERAND_EXPECTED, 12(SP)	5273	
	03	0C	AE	E8	0010C		BLBS	12(SP), -16\$		
			00FB	31	00110	15\$:	BRW	28\$		
	50	00000000'	FF	9A	00113	16\$:	MOVZBL	@CHARPTR, R0		
	EE	00000000'	EF40	E9	0011A		BLBC	CHARTBL+2(R0), 15\$		
		00000000'	EF	D6	00122	17\$:	INCL	CHARPTR	5276	
	50	00000000'	FF	9A	00128		MOVZBL	@CHARPTR, R0	5277	
		00000000'	EF40	DF	0012F		PUSHAL	CHARTBL+2(R0)		
E8	9E		01	E0	00136		BBS	#1, @ (SP)+, 17\$		
	50	00000000G	00	E9	0013A		BLBC	DBG\$GB SET BREAK_FLAG, 20\$	5285	
	50	00000000'	EF	D0	00141		MOVL	CHARPTR, R0	5288	
44	8F		60	91	00148		CMPB	(R0), #68		
			1C	12	0014C		BNEQ	19\$		
4F	8F	01	A0	91	0014E		CMPB	1(R0), #79	5289	
			15	12	00153		BNEQ	19\$		
	20	02	A0	91	00155		CMPB	2(R0), #32	5290	
			06	13	00159		BEQL	18\$		
	28	02	A0	91	0015B		CMPB	2(R0), #40		
			09	12	0015F		BNEQ	19\$		
	51	00000000'	EF	9E	00161	18\$:	MOVAB	CURLOC_TOKEN, R1	5292	
			7C	11	00168		BRB	24\$		
57	8F		60	91	0016A	19\$:	CMPB	(R0), #87	5293	
			21	12	0016E		BNEQ	20\$		
48	8F	01	A0	91	00170		CMPB	1(R0), #72	5294	
			1A	12	00175		BNEQ	20\$		
45	8F	02	A0	91	00177		CMPB	2(R0), #69	5295	
			13	12	0017C		BNEQ	20\$		
4E	8F	03	A0	91	0017E		CMPB	3(R0), #78	5296	
			0C	12	00183		BNEQ	20\$		
	20	04	A0	91	00185		CMPB	4(R0), #32	5297	
			74	13	00189		BEQL	26\$		
	28	04	A0	91	0018B		CMPB	4(R0), #40		
			6E	13	0018F		BEQL	26\$		
	50	00000000'	FF	9A	00191	20\$:	MOVZBL	@CHARPTR, R0	5331	
	67	00000000'	EF40	E8	00198		BLBS	CHARTBL(R0), 27\$		
		00000000'	EF40	DF	001A0		PUSHAL	CHARTBL+1(R0)	5332	
	9E		01	E0	001A7		BBS	#1, @ (SP)+, 27\$		
5C	25		50	91	001AB		CMPB	R0, #37	5333	
			0D	12	001AE		BNEQ	21\$		
	07	00000000G	00	91	001B0		CMPB	DBG\$GB_LANGUAGE, #7	5334	
			4E	12	001B7		BNEQ	27\$		
5C	4A	08	AC	E8	001B9		BLBS	ADDRESS_EXPRESSION, 27\$		
	8F		66	91	001BD	21\$:	CMPB	(STARTPTR), #92	5337	
			09	12	001C1		BNEQ	22\$		
	51	00000000'	EF	9E	001C3		MOVAB	CURVAL_TOKEN, R1		
			1A	11	001CA		BRB	24\$		
5E	8F		66	91	001CC	22\$:	CMPB	(STARTPTR), #94	5338	
			18	12	001D0		BNEQ	25\$		
	05	00000000G	00	91	001D2		CMPB	DBG\$GB_LANGUAGE, #5	5339	

			04	12	001D9	BNEQ	23\$		
			AC	E9	001DB	BLBC	ADDRESS_EXPRESSION, 25\$		
			EF	9E	001DF	MOVAB	PREVLOC_TOKEN, R1	5341	
			51	00	001E6	MOVL	R1, R0		
			50	04	001E9	RET			
			2E	66	91	001EA	25\$: CMPB (STARTPTR), #46	5342	
			18	12	001ED	BNEQ	27\$		
			28	50	91	001EF	CMPB R0, #40	5343	
			13	13	001F2	BEQL	27\$		
			2E	50	91	001F4	CMPB R0, #46	5344	
			0E	13	001F7	BEQL	27\$		
5C			50	91	001F9	CMPB	R0, #92	5345	
			08	13	001FD	BEQL	27\$		
			50	9E	001FF	26\$: MOVAB	CURLOC_TOKEN, R0	5347	
			EF	04	00206	RET			
			56	00	00207	27\$: MOVL	STARTPTR, CHARPTR	5351	
			50	00	0020E	28\$: MOVL	CHARPTR, R0	5359	
			25	60	91	00215	CMPB (R0), #37		
			03	13	00218	BEQL	30\$		
			02C8	31	0021A	29\$: BRW	68\$		
			AE	E9	0021D	30\$: BLBC	12(SP), 29\$	5360	
			01	9A	00221	MOVZBL	1(R0), R0	5361	
			ED	E9	00225	BLBC	CHARTBL+1(R0), 29\$		
			00000000'	EF	D6	0022D	INCL	CHARPTR	5368
			00000000'	01	00	00233	MOVL	#1, TOKENLEN	5369
			59	25	90	00236	MOVB	#37, TOKENBUFFER+1	5370
15			AE	FF	9A	0023A	31\$: MOVZBL	@CHARPTR, R0	5371
			50	00000000'	EF	40	00241	BLBS	CHARTBL+1(R0), 32\$
			0B	00000000'	EF	40	00249	PUSHAL	CHARTBL+1(R0)
			00000000'	01	E1	00250	BBC	#1, @ (SP)+, 33\$	5372
OF			9E	59	D6	00254	32\$: INCL	TOKENLEN	5375
			14	AE49	50	90	00256	MOVB	R0, TOKENBUFFER[TOKENLEN]
				00000000'	EF	D6	0025B	INCL	CHARPTR
					D7	11	00261	BRB	31\$
			14	AE	59	90	00263	33\$: MOVB	TOKENLEN, TOKENBUFFER
					55	D4	00267	CLRL	INDEX
			54		01	CE	00269	MNEGL	#1, I
					23	11	0026C	BRB	35\$
			50	00000000'	EF	9E	0026E	34\$: MOVAB	TABLEBASE, R0
			50	00000000'	EF	44	C1	00275	ADDL3
			50	01	AB	9A	0027E	MOVZBL	1(NAMEPTR), R0
			00	02	AB	2D	00282	CMPC5	R0, 2(NAMEPTR), #0, TOKENLEN, TOKENBUFFER+1
				15	AE		00288		
					05	12	0028A	BNEQ	35\$
			55		6B	9A	0028C	MOVZBL	(NAMEPTR), INDEX
					08	11	0028F	BRB	36\$
					EF	F2	00291	35\$: AOBLSS	PERCENT_TABLE-4, 1, 34\$
			54	00000000'	55	CF	00299	36\$: CASEL	INDEX, #0, #8
			00				0029D	37\$: .WORD	67\$-37\$,-
			0241				002A5		38\$-37\$,-
			0221				002AD		38\$-37\$,-
			0B39						54\$-37\$,-
									63\$-37\$,-
									64\$-37\$,-
									65\$-37\$,-
									66\$-37\$,-
									201\$-37\$

019C 0239

05 08 002A 0231

002A 0229

0241 0221 0B39

		00000000'	EF	9F	002AF	PUSHAB	P.AXC	5611	
			01	DD	002B5	PUSHL	#1		
		00028362	8F	DD	002B7	PUSHL	#164706		
	00		03	FB	002BD	CALLS	#3, LIB\$SIGNAL		
			0217	31	002C4	BRW	67\$		
			57	D4	002C7	CLRL	R7	5424	
	01		55	D1	002C9	CMPL	INDEX, #1		
			17	12	002CC	BNEQ	39\$		
			57	D6	002CE	INCL	R7		
15	AE	00000000'	EF	06	28	002D0	MOVCS	#6, P.AXA, TOKENBUFFER+1	5427
			59	06	DD	002D9	MOVL	#6, TOKENLEN	5428
		000289F2	8F	DD	002DC	MOVL	#166386, ERRORMSG	5429	
			13	11	002E3	BRB	40\$	5424	
15	AE	00000000'	EF	07	28	002E5	MOVCS	#7, P.AXB, TOKENBUFFER+1	5434
			59	07	DD	002EE	MOVL	#7, TOKENLEN	5435
		000289EA	8F	DD	002F1	MOVL	#166378, ERRORMSG	5436	
		00000000'	FF	9A	002F8	MOVZBL	@CHARPTR, R0	5443	
		00000000'	EF	40	DF	002FF	PUSHAL	CHARTBL+2[R0]	
09			01	E0	00306	BBS	#1, @ (SP)+, 41\$		
			52	DD	0030A	PUSHL	ERRORMSG		
	00		01	FB	0030C	CALLS	#1, LIB\$SIGNAL		
		00000000'	FF	9A	00313	MOVZBL	@CHARPTR, R0	5444	
		00000000'	EF	40	DF	0031A	PUSHAL	CHARTBL+2[R0]	
08			01	E1	00321	BBC	#1, @ (SP)+, 42\$		
		00000000'	EF	D6	00325	INCL	CHARPTR	5445	
			E6	11	0032B	BRB	41\$		
	50	00000000'	FF	9A	0032D	MOVZBL	@CHARPTR, R0	5450	
		00000000'	EF	40	DF	00334	PUSHAL	CHARTBL+1[R0]	
09			01	E0	0033B	BBS	#1, @ (SP)+, 43\$		
			52	DD	0033F	PUSHL	ERRORMSG	5452	
	00		01	FB	00341	CALLS	#1, LIB\$SIGNAL		
		00000000'	EF	DD	00348	MOVL	CHARPTR, R0	5454	
			60	91	0034F	CMPB	(R0), #48		
			17	12	00352	BNEQ	44\$		
	50	01	A0	9A	00354	MOVZBL	1(R0), R0	5455	
		00000000'	EF	40	DF	00358	PUSHAL	CHARTBL+1[R0]	
08			01	E1	0035F	BBC	#1, @ (SP)+, 44\$		
		00000000'	EF	D6	00363	INCL	CHARPTR	5457	
			DD	11	00369	BRB	43\$		
	50	00000000'	FF	9A	0036B	MOVZBL	@CHARPTR, R0	5463	
		00000000'	EF	40	DF	00372	PUSHAL	CHARTBL+1[R0]	
25			01	E1	00379	BBC	#1, @ (SP)+, 46\$		
		000000FF	8F	59	D1	0037D	CMPL	TOKENLEN, #255	5465
			09	19	00384	BLSS	45\$		
			52	DD	00386	PUSHL	ERRORMSG		
	00		01	FB	00388	CALLS	#1, LIB\$SIGNAL		
			59	D6	0038F	INCL	TOKENLEN	5466	
	14	AE49	00000000'	FF	90	00391	MOVB	@CHARPTR, TOKENBUFFER[TOKENLEN]	5467
			00000000'	EF	D6	0039A	INCL	CHARPTR	5468
			C9	11	003A0	BRB	44\$	5463	
	03		57	E8	003A2	BLBS	R7, 48\$	5476	
			010F	31	003A5	BRW	61\$		
	2E	00000000'	FF	91	003A8	CMPB	@CHARPTR, #46		
			F4	12	003AF	BNEQ	47\$		
	50	00000000'	EF	DD	003B1	MOVL	CHARPTR, R0	5483	
			50	D6	003B8	INCL	R0		
	50		60	9A	003BA	MOVZBL	(R0), R0		

09	000000FF	9E	00000000	'EF40	DF	003BD	PUSHAL	CHARTBL+1[RO]	
		8F		01	E1	003C4	BBC	#1, a(SP)+, 49\$	5484
				59	D1	003C8	CMPL	TOKENLEN, #255	
				09	19	003CF	BLSS	50\$	5486
	00000000G	00		52	DD	003D1	PUSHL	ERRORMSG	
				01	FB	003D3	CALLS	#1, LIB\$SIGNAL	5488
	14	AE49		59	D6	003DA	INCL	TOKENLEN	5489
				2E	90	003DC	MOVB	#46, TOKENBUFFER[TOKENLEN]	5490
			00000000'	EF	D6	003E1	INCL	CHARPTR	5495
		50	00000000'	EF	D0	003E7	MOVL	CHARPTR, RO	
		30		60	91	003EE	CMPL	(RO), #48	
				0F	12	003F1	BNEQ	52\$	
		50	01	A0	9A	003F3	MOVZBL	1(RO), RO	5496
DF			00000000'	'EF40	DF	003F7	PUSHAL	CHARTBL+1[RO]	
		9E		01	E0	003FE	BBS	#1, a(SP)+, 51\$	
		50	00000000'	FF	9A	00402	MOVZBL	@CHARPTR, RO	5503
			00000000'	'EF40	DF	00409	PUSHAL	CHARTBL+1[RO]	
91	000000FF	9E		01	E1	00410	BBC	#1, a(SP)+, 47\$	
		8F		59	D1	00414	CMPL	TOKENLEN, #255	5505
				09	19	0041B	BLSS	53\$	
				52	DD	0041D	PUSHL	ERRORMSG	
	00000000G	00		01	FB	0041F	CALLS	#1, LIB\$SIGNAL	5506
				59	D6	00426	INCL	TOKENLEN	5507
	14	AE49	00000000'	FF	90	00428	MOVB	@CHARPTR, TOKENBUFFER[TOKENLEN]	5508
			00000000'	EF	D6	00431	INCL	CHARPTR	5503
				C9	11	00437	BRB	52\$	5533
		50	00000000'	FF	9A	00439	MOVZBL	@CHARPTR, RO	
			00000000'	'EF40	DF	00440	PUSHAL	CHARTBL+2[RO]	
08		9E		01	E1	00447	BBC	#1, a(SP)+, 55\$	
			00000000'	EF	D6	0044B	INCL	CHARPTR	5534
				E6	11	00451	BRB	54\$	
		56	00000000'	EF	D0	00453	MOVL	CHARPTR, STARTPTR	5536
		50	00000000'	FF	9A	0045A	MOVZBL	@CHARPTR, RO	5542
			00000000'	'EF40	DF	00461	PUSHAL	CHARTBL+1[RO]	
0D		9E		02	E1	00468	BBC	#2, a(SP)+, 56\$	
			10	AE	9F	0046C	PUSHAB	TOKEN TYPE	5544
			18	AE	9F	0046F	PUSHAB	TOKENBUFFER	
	0000V	CF		02	FB	00472	CALLS	#2, SCAN_QUOTED_STRING	
				42	11	00477	BRB	62\$	
		50	00000000'	FF	9A	00479	MOVZBL	@CHARPTR, RO	5553
		50	00000000'	'EF40	DE	00480	MOVAL	CHARTBL[RO], RO	
		08		60	E8	00488	BLBS	(RO), 57\$	
04		60		01	E0	0048B	BBS	#1, (RO), 57\$	5554
08		60		02	E1	0048F	BBC	#2, (RO), 58\$	5555
			00000000'	EF	D6	00493	INCL	CHARPTR	5557
				DE	11	00499	BRB	56\$	
59	00000000'	EF		56	C3	0049B	SUBL3	STARTPTR, CHARPTR, TOKENLEN	5559
				0D	12	004A3	BNEQ	60\$	5560
			000289BA	8F	DD	004A5	PUSHL	#166330	
15	AE	00000000G	00	01	FB	004AB	CALLS	#1, LIB\$SIGNAL	
			66	59	28	004B2	MOVC3	TOKENLEN, (STARTPTR), TOKENBUFFER+1	5561
		14	AE	59	90	004B7	MOVB	TOKENLEN, TOKENBUFFER	5562
				0918	31	004BB	BRW	201\$	5577
		50	00000000'	EF	9E	004BE	MOVAB	RADIX_OP_DEC, RO	5584
				04	04	004C5	RET		
		50	00000000'	EF	9E	004C6	MOVAB	RADIX_OP_HEX, RO	5591
				04	04	004CD	RET		

	50	00000000'	EF	9E	004CE	65\$:	MOVAB	RADIX_OP_OCT, R0	5598	
				04	004D5		RET			
	50	00000000'	EF	9E	004D6	66\$:	MOVAB	RADIX_OP_BIN, R0	5605	
				04	004DD		RET			
	00000000'	EF	56	D0	004DE	67\$:	MOVL	STARTPTR, CHARPTR	5621	
	50	00000000'	FF	9A	004E5	68\$:	MOVZBL	@CHARPTR, R0	5634	
		00000000'	EF	40	DF	004EC	PUSHAL	CHARTBL+1(R0)		
14	9E		02	E1	004F3		BBC	#2, @ (SP)+, 69\$		
		10	AE	9F	004F7		PUSHAB	TOKEN_TYPE	5637	
		18	AE	9F	004FA		PUSHAB	TOKENBUFFER		
	0000V	CF	02	FB	004FD		CALLS	#2, SCAN_QUOTED_STRING		
		14	AE	9F	00502		PUSHAB	TOKENBUFFER	5638	
		14	AE	DD	00505		PUSHL	TOKEN_TYPE		
			08	D0	31	00508	BRW	202\$		
	50	00000000'	FF	9A	0050B	69\$:	MOVZBL	@CHARPTR, R0	5655	
		00000000'	EF	40	DF	00512	PUSHAL	CHARTBL(R0)		
03	9E		03	E0	00519		BBS	#3, @ (SP)+, 71\$		
			01	D8	31	0051D	BRW	107\$		
	05	08	AC	E9	00520	70\$:	BLBC	ADDRESS_EXPRESSION, 72\$	5656	
	2E		50	91	00524	71\$:	CMPB	R0, #46		
			F4	13	00527		BEQL	70\$		
			58	D4	00529	72\$:	CLRL	STATE_INDEX	5669	
	09	00000000G	00	91	0052B		CMPB	DBG\$GB_LANGUAGE, #9	5670	
			0C	12	00532		BNEQ	73\$		
	0A	00000000'	EF	D1	00534		CMPL	EXPRESSION_RADIX, #10	5671	
			03	13	0053B		BEQL	73\$		
	58		1F	D0	0053D		MOVL	#31, STATE_INDEX	5673	
	57		04	D0	00540	73\$:	MOVL	#4, NUMBER_KIND	5675	
	50	00000000'	EF	D0	00543		MOVL	EXPRESSION_RADIX, R0	5676	
	10		50	D1	0054A		CMPL	R0, #16		
			03	12	0054D		BNEQ	74\$		
	57		05	D0	0054F		MOVL	#5, NUMBER_KIND	5678	
	08		50	D1	00552	74\$:	CMPL	R0, #8	5680	
			03	12	00555		BNEQ	75\$		
	57		0B	D0	00557		MOVL	#11, NUMBER_KIND	5682	
	02		50	D1	0055A	75\$:	CMPL	R0, #2	5684	
			03	12	0055D		BNEQ	76\$		
	57		0A	D0	0055F		MOVL	#10, NUMBER_KIND	5686	
	5A		57	D0	00562	76\$:	MOVL	NUMBER_KIND, BACKUP_NUMBER_KIND	5688	
	52	00000000'	EF	D0	00565	77\$:	MOVL	CHARPTR, R2	5703	
	50		62	9A	0056C		MOVZBL	(R2), R0		
	53	00000000'	EF	40	DE	0056F	MOVAL	CHARTBL(R0), R3		
04	04		04	EF	00577		EXTZV	#4, #4, (R3), CLASS		
	09	00000000G	00	91	0057D		CMPB	DBG\$GB_LANGUAGE, #9	5704	
			0E	12	00584		BNEQ	78\$		
	2E		50	91	00586		CMPB	R0, #46	5706	
			09	12	00589		BNEQ	78\$		
	2E	01	A2	91	0058B		CMPB	1(R2), #46		
			03	12	0058F		BNEQ	78\$		
		04	AE	D4	00591		CLRL	CLASS	5708	
		00000000'	FF	48	DF	00594	78\$:	PUSHAL	@STATE_TABLE[STATE_INDEX]	5717
	50		9E	9A	0059B		MOVZBL	@ (SP)+, R0		
			0A	13	0059E		BEQL	79\$		
04	AE		50	D1	005A0		CMPL	R0, CLASS	5719	
			04	13	005A4		BEQL	79\$		
			58	D6	005A6		INCL	STATE_INDEX	5721	
			EA	11	005AB		BRB	7B\$		

08	AE	9E	08	00000000'FF48	DF	005AA	79\$:	PUSHAL	@STATE_TABLE[STATE_INDEX]	5723
					EF	005B1		EXTZV	#8, #8, @ (SP)+, ACTION	
	5B	9E	10	00000000'FF48	DF	005B7		PUSHAL	@STATE_TABLE[STATE_INDEX]	5724
		11	01		EF	005BE		EXTZV	#16, #16, @ (SP)+, STATE_INDEX	
0045			08	0B	CF	005C3		CASEL	ACTION, #1, #17	5730
008F		0040	0127			005C8	80\$:	.WORD	106\$-80\$,-	
00D2		0059	004A			005D0			82\$-80\$,-	
00B3		0024	00AD			005D8			83\$-80\$,-	
		0072	00C7			005E0			84\$-80\$,-	
			0127			005E8			85\$-80\$,-	
									86\$-80\$,-	
									88\$-80\$,-	
									94\$-80\$,-	
									96\$-80\$,-	
									101\$-80\$,-	
									81\$-80\$,-	
									100\$-80\$,-	
									98\$-80\$,-	
									89\$-80\$,-	
									90\$-80\$,-	
									97\$-80\$,-	
									106\$-80\$,-	
									87\$-80\$	
				00000000'	EF	9F 005EC	81\$:	PUSHAB	P, AXD	5944
					01	DD 005F2		PUSHL	#1	
				00028362	8F	DD 005F4		PUSHL	#164706	
00000000G	00		03		FB	005FA		CALLS	#3, LIB\$SIGNAL	
			51		11	00601		BRB	93\$	
	6E		52		DD	00603	82\$:	MOVL	R2, BACKUP_DIGIT_PTR	5745
			4C		11	00606		BRB	93\$	
	57		06		DD	00608	83\$:	MOVL	#6, NUMBER_KIND	5752
			47		11	0060B		BRB	93\$	
	6E		52		DD	0060D	84\$:	MOVL	R2, BACKUP_DIGIT_PTR	5761
			3F		11	00610		BRB	92\$	5762
	57		07		DD	00612	85\$:	MOVL	#7, NUMBER_KIND	5769
			3D		11	00615		BRB	93\$	
	57		08		DD	00617	86\$:	MOVL	#8, NUMBER_KIND	5775
			38		11	0061A		BRB	93\$	
	57		0F		DD	0061C	87\$:	MOVL	#15, NUMBER_KIND	5781
			33		11	0061F		BRB	93\$	
	57		09		DD	00621	88\$:	MOVL	#9, NUMBER_KIND	5787
			2E		11	00624		BRB	93\$	
	6E		52		DD	00626	89\$:	MOVL	R2, BACKUP_DIGIT_PTR	5796
05			57		D1	00629		CMPL	NUMBER_KIND, #5	5797
			68		13	0062C		BEQL	99\$	
0B			57		D1	0062E		CMPL	NUMBER_KIND, #11	5798
			63		13	00631		BEQL	99\$	
0A			57		D1	00633		CMPL	NUMBER_KIND, #10	5799
			5E		13	00636		BEQL	99\$	
			12		11	00638		BRB	91\$	5801
	6E		52		DD	0063A	90\$:	MOVL	R2, BACKUP_DIGIT_PTR	5811
05			57		D1	0063D		CMPL	NUMBER_KIND, #5	5812
			0F		13	00640		BEQL	92\$	
0B			57		D1	00642		CMPL	NUMBER_KIND, #11	5813
			0A		13	00645		BEQL	92\$	
0A			57		D1	00647		CMPL	NUMBER_KIND, #10	5814
			05		13	0064A		BEQL	92\$	

	5A	0E	D0	0064C	91\$:	MOVL	#14, BACKUP_NUMBER_KIND	5816		
		03	11	0064F		BRB	93\$			
	5A	06	D0	00651	92\$:	MOVL	#6, BACKUP_NUMBER_KIND	5818		
		0098	31	00654	93\$:	BRW	106\$	5730		
00000000'	EF	5A	D0	00657	94\$:	MOVL	BACKUP_NUMBER_KIND, NUMBER_KIND	5831		
	59	01	C1	0065A		ADDL3	#1, BACKUP_DIGIT_PTR, CHARPTR	5832		
		6E	C3	00662		SUBL3	STARTPTR, CHARPTR, TOKENLEN	5833		
	EF	56	D1	0066A	95\$:	CMPL	TOKENLEN, #255	5834		
		8F	5E	14	00671	BGTR	104\$			
			69	11	00673	BRB	105\$	5835		
	59	52	56	C3	00675	96\$:	SUBL3	STARTPTR, R2, TOKENLEN	5847	
			EF	11	00679	BRB	95\$	5848		
	05	57	D1	0067B	97\$:	CMPL	NUMBER_KIND, #5	5861		
			F5	13	0067E	BEQL	96\$			
	0B	57	D1	00680		CMPL	NUMBER_KIND, #11	5862		
			F0	13	00683	BEQL	96\$			
	0A	57	D1	00685		CMPL	NUMBER_KIND, #10	5863		
			EB	13	00688	BEQL	96\$			
	57	0E	D0	0068A		MOVL	#14, NUMBER_KIND	5865		
			E6	11	0068D	BRB	96\$	5867		
	10	00000000'	EF	D1	0068F	98\$:	CMPL	EXPRESSION_RADIX, #16	5889	
			57	13	00696	99\$:	BEQL	106\$		
			0B	11	00698	BRB	101\$	5892		
	08	63	E8	0069A	100\$:	BLBS	(R3), 101\$	5909		
			01	E0	0069D	BBS	#1, (R3), 101\$	5910		
04	63		02	E1	006A1	BBC	#2, (R3), 102\$	5911		
09	63		56	D0	006A5	101\$:	MOVL	STARTPTR, CHARPTR	5914	
	00000000'	EF	4A	11	006AC	BRB	107\$	5913		
			05	57	D1	006AE	102\$:	CMPL	NUMBER_KIND, #5	5918
			0D	13	006B1	BEQL	103\$			
	0B	57	D1	006B3		CMPL	NUMBER_KIND, #11	5919		
			08	13	006B6	BEQL	103\$			
	0A	57	D1	006B8		CMPL	NUMBER_KIND, #10	5920		
			03	13	006BB	BEQL	103\$			
	57	0E	D0	006BD		MOVL	#14, NUMBER_KIND	5922		
			56	C3	006C0	103\$:	SUBL3	STARTPTR, CHARPTR, TOKENLEN	5924	
	59	00000000'	59	D1	006C8		CMPL	TOKENLEN, #255	5925	
		0000000FF	8F	0D	15	006CF	BLEQ	105\$		
			000289C2	8F	DD	006D1	104\$:	PUSHL	#166338	
				01	FB	006D7	CALLS	#1, LIB\$SIGNAL		
	00000000G	00	59	90	006DE	105\$:	MOVB	TOKENLEN, TOKENBUFFER	5926	
		14	59	28	006E2		MOVC3	TOKENLEN, (STARTPTR), TOKENBUFFER+1	5927	
15	AE		14	AE	9F	006E7	PUSHAB	TOKENBUFFER	5928	
				57	DD	006EA	PUSHL	NUMBER_KIND		
				06EC	31	006EC	BRW	202\$		
		00000000'	EF	D6	006EF	106\$:	INCL	CHARPTR	5951	
			FE6D	31	006F5		BRW	77\$	5695	
	57	00000000'	EF	D0	006F8	107\$:	MOVL	CHARPTR, R7	5963	
	50		67	9A	006FF		MOVZBL	(R7), R0		
	03	00000000'	EF40	E8	00702		BLBS	CHAR[BL[R0], 108\$		
			0147	31	0070A		BRW	126\$		
			57	D7	0070D	108\$:	DECL	ENDPTR	5974	
	51	00000000'	EF	D0	0070F		MOVL	CHARPTR, R1	5977	
			50	61	9A	00716	109\$:	MOVZBL	(R1), R0	
		00000000'	EF40	DF	00719		PUSHAL	CHAR[BL[R0]		
	OE			02	E1	00720	BBC	#2, @ (SP)+, 110\$		
			57	51	D0	00724	MOVL	R1, ENDPTR	5980	

26	9E	00000000'	EF40	DF	00727	PUSHAL	CHARTBL[R0]	5981	
			01	E1	0072E	BBC	#1, @ (SP)+, 111\$		
	51	00000000'	EF	D6	00732	INCL	CHARPTR	5984	
	50	00000000'	61	9A	0073F	MOVL	CHARPTR, R1	5985	
						MOVZBL	(R1), R0		
C9	9E	00000000'	EF40	DF	00742	PUSHAL	CHARTBL[R0]		
			01	E0	00749	BBS	#1, @ (SP)+, 109\$		
BE	9E	00000000'	EF40	DF	0074D	PUSHAL	CHARTBL[R0]	5986	
			02	E0	00754	BBS	#2, @ (SP)+, 109\$		
59	EF	00000000'	01	A7	9E	00758	111\$: MOVAB	1(R7), CHARPTR	5990
	5F	00000000'		56	C3	00760	SUBL3	STARTPTR, CHARPTR, TOKENLEN	5995
	8F	000000FF		59	D1	00768	CMPL	TOKENLEN, #255	5996
				0D	15	0076F	BLEQ	112\$	
		00028982		8F	DD	00771	PUSHL	#166274	
15	AE	00000000G	00	01	FB	00777	CALLS	#1, LIB\$SIGNAL	
	66			59	28	0077E	112\$: MOVAB	TOKENLEN, (STARTPTR), TOKENBUFFER+1	5997
	14			59	90	00783	MOVB	TOKENLEN, TOKENBUFFER	5998
	3B	00000000'		5F	E9	00787	BLBC	CASING SIGNIFICANT, 115\$	6010
	00	00000000G		56	D1	0078E	CMPL	STARTPTR, DBG\$GL_UPCASE_COMMAND_PTR	6013
				09	19	00795	BLSS	113\$	
	00	00000000G		56	D1	00797	CMPL	STARTPTR, DBG\$GL_UPCASE_COMMAND_PTR+4	6014
				15	15	0079E	BLEQ	114\$	
		00000000'		EF	9F	007A0	113\$: PUSHAB	P.AXE	6016
				01	DD	007A6	PUSHL	#1	
		00028362		8F	DD	007AB	PUSHL	#164706	
50	00	00000000G	00	03	FB	007AE	CALLS	#3, LIB\$SIGNAL	
	56	00000000G	00	C3	007B5	114\$: SUBL3	DBG\$GL_UPCASE COMMAND PTR, STARTPTR, R0	6018	
	50	00000000G	00	C0	007BD	ADDL2	DBG\$GL_ORIG COMMAND PTR, NEW STARTPTR	6019	
15	AE		60	59	28	007C4	115\$: MOVAB	TOKENLEN, (NEW STARTPTR), TOKENBUFFER+1	6020
				5A	D4	007C9	CLRL	BEST_TOKEN_FOUND	6028
	55	00000000'		EF	D0	007CB	MOVL	IDENT_OPERATOR_TABLE, R5	6029
	54			01	CE	007D2	MNEGL	#1, I	
				32	11	007D5	BRB	121\$	
	50	00000000'		EF	9E	007D7	116\$: MOVAB	TABLEBASE, R0	6031
58	50		6544	C1	007DE	ADDL3	(R5)[I], R0, TOKEN		
	50	0C	A8	9A	007E3	MOVZBL	12(TOKEN), R0	6032	
59	00	0D	A8	2D	007E7	CMPC5	R0, 13(TOKEN), #0, TOKENLEN, TOKENBUFFER+1	6033	
			15	AE					
				18	12	007EF	BNEQ	121\$	
	0C	0C	AE	E9	007F1	BLBC	12(SP), 119\$	6036	
	02		68	91	007F5	CMPL	(TOKEN), #2	6037	
			03	12	007F8	BNEQ	118\$		
		0549	31	007FA	117\$: BRW		193\$		
	05	0C	AE	E8	007FD	118\$: BLBS	12(SP), 120\$	6038	
	02		68	91	00801	119\$: CMPL	(TOKEN), #2	6039	
			F4	12	00804	BNEQ	117\$		
	5A		58	D0	00806	120\$: MOVL	TOKEN, BEST_TOKEN_FOUND	6044	
C9	54	FC	A5	F2	00809	121\$: AOBLS	-4(R5), I, T16\$	6029	
			5A	D5	0080E	TSTL	BEST_TOKEN_FOUND	6050	
			36	12	00810	BNEQ	124\$		
			5A	D4	00812	CLRL	BEST_TOKEN_FOUND	6057	
	55	00000000'		EF	D0	00814	MOVL	BIF_TABLE, R5	6058
	54			01	CE	0081B	MNEGL	#1, I	
				21	11	0081E	BRB	123\$	
	50	00000000'		EF	9E	00820	122\$: MOVAB	TABLEBASE, R0	6060
58	50		6544	C1	00827	ADDL3	(R5)[I], R0, TOKEN		
	50	0B	A8	9A	0082C	MOVZBL	8(TOKEN), R0	6061	

59	00	09	A8	15	50	2D	00830	CMPC5	RO, 9(TOKEN), #0, TOKENLEN, TOKENBUFFER+1	6062	
				07	AE	12	00836	BNEQ	123\$		
			BC	0C	AE	E8	0083A	BLBS	12(SP), 117\$	6064	
			5A		58	D0	0083E	MOVL	TOKEN, BEST_TOKEN_FOUND	6068	
	DA		54	FC	A5	F2	00841	AOBLS	-4(R5), 1, T22\$	6058	
					5A	D5	00846	TSTL	BEST_TOKEN_FOUND	6071	
					03	13	00848	BEQL	125\$		
				01BF	31	0084A	BRW	154\$			
				59	D5	0084D	125\$:	TSTL	TOKENLEN	6078	
				03	13	0084F	BEQL	126\$			
				0582	31	00851	BRW	201\$			
		50	00000000'	FF	9A	00854	126\$:	MOVZBL	@CHARPTR, RO	6088	
			00000000'	EF40	DF	0085B	PUSHAL	CHARTBL+1[RO]			
	03	9E		03	E0	00862	BBS	#3, a(SP)+, 127\$			
				00F1	31	00866	BRW	143\$			
		50	00000000'	FF	9A	00869	127\$:	MOVZBL	@CHARPTR, RO	6100	
			00000000'	EF40	DF	00870	PUSHAL	CHARTBL+1[RO]			
	08	9E		04	E1	00877	BBC	#4, a(SP)+, 128\$			
			00000000'	EF	D6	0087B	INCL	CHARPTR		6101	
				E6	11	00881	BRB	127\$			
		50	00000000'	FF	9A	00883	128\$:	MOVZBL	@CHARPTR, RO	6103	
			00000000'	EF40	DF	0088A	PUSHAL	CHARTBL+1[RO]			
	08	9E		D5	E1	00891	BBC	#5, a(SP)+, 129\$			
			00000000'	EF	D6	00895	INCL	CHARPTR		6104	
				E6	11	0089B	BRB	128\$			
		50	00000000'	FF	9A	0089D	129\$:	MOVZBL	@CHARPTR, RO	6106	
			00000000'	EF40	DF	008A4	PUSHAL	CHARTBL+1[RO]			
	08	9E		C6	E1	008AB	BBC	#6, a(SP)+, 130\$			
			00000000'	EF	D6	008AF	INCL	CHARPTR		6107	
				E6	11	008B5	BRB	129\$			
		56	00000000'	EF	D1	008B7	130\$:	CMPL	CHARPTR, STARTPTR	6109	
				06	12	008BE	BNEQ	131\$			
			00000000'	EF	D6	008C0	INCL	CHARPTR			
	59	00000000'	EF	56	C3	008C6	131\$:	SUBL3	STARTPTR, CHARPTR, TOKENLEN	6114	
		000000FF	8F	59	D1	008CE	CMPL	TOKENLEN, #255		6115	
				DD	15	008D5	BLEQ	132\$			
			000289A2	8F	DD	008D7	PUSHL	#166306			
		00000000G	00	01	FB	008DD	CALLS	#1, LIB\$SIGNAL			
		14	AE	59	90	008E4	132\$:	MOVB	TOKENLEN, TOKENBUFFER	6116	
15	AE		66	59	28	008E8	MOV3	TOKENLEN, (STARTPTR), TOKENBUFFER+1		6117	
				5A	D4	008ED	CLRL	BEST_TOKEN_FOUND		6126	
		55	00000000'	EF	D0	008EF	MOVL	OPCHAR_OPERATOR_TABLE, R5		6127	
		54		01	CE	008F6	MNEGL	#1, 1			
				3A	11	008F9	BRB	139\$			
		50	00000000'	EF	9E	008FB	133\$:	MOVAB	TABLEBASE, RO	6129	
	58			6544	C1	00902	ADDL3	(R5)[1], RO, TOKEN			
		04		01	A8	E8	00907	BLBS	1(TOKEN), 134\$	6130	
		26		08	AC	E8	0090B	BLBS	ADDRESS_EXPRESSION, 139\$		
		50		0C	A8	9A	0090F	134\$:	MOVZBL	12(TOKEN), RO	6131
59	00	0D	A8	15	50	2D	00913	CMPC5	RO, 13(TOKEN), #0, TOKENLEN, TOKENBUFFER+1	6132	
					AE		00919				
				18	12	0091B	BNEQ	139\$			
		0C		0C	AE	E9	0091D	BLBC	12(SP), 137\$	6135	
		02		68	91	00921	CMPB	(TOKEN), #2		6136	
				03	12	00924	BNEQ	136\$			
				041D	31	00926	135\$:	BRW	193\$		

		05	OC	AE	E8	00929	136\$:	BLBS	12(SP), 138\$	6137	
		02		68	91	0092D	137\$:	CMPB	(TOKEN), #2	6138	
				F4	12	00930		BNEQ	135\$		
		5A		58	D0	00932	138\$:	MOVL	TOKEN, BEST_TOKEN_FOUND	6143	
C1		54	FC	A5	F2	00935	139\$:	AOBLSS	-4(R5), 1, T33\$	6127	
				5A	D5	0093A		TSTL	BEST_TOKEN_FOUND	6155	
				15	13	0093C		BEQL	142\$		
		03	0B	AC	E8	0093E		BLBS	ADDRESS_EXPRESSION, 141\$	6156	
				00C7	31	00942	140\$:	BRW	154\$		
		50		66	9A	00945	141\$:	MOVZBL	(STARTPTR), R0	6157	
				00000000'	EF40	DF	00948	PUSHAL	CHARTBL+1[R0]		
					9E	95	0094F	TSTB	@(SP)+		
					EF	18	00951	BGEQ	140\$		
	00000000'	EF		56	D0	00953	142\$:	MOVL	STARTPTR, CHARPTR	6165	
		03	0B	AC	E8	0095A	143\$:	BLBS	ADDRESS_EXPRESSION, 145\$	6176	
				00B6	31	0095E	144\$:	BRW	156\$		
		50		00000000'	FF	9A	00961	145\$:	MOVZBL	@CHARPTR, R0	
				00000000'	EF40	DF	00968	PUSHAL	CHARTBL+1[R0]		
					9E	95	0096F	TSTB	@(SP)+		
					EB	18	00971	BGEQ	144\$		
					51	D4	00973	CLRL	R1	6182	
		07	00000000G	00	91	00975		CMPB	DBG\$GB_LANGUAGE, #7		
				22	12	0097C		BNEQ	146\$		
				51	D6	0097E		INCL	R1		
		2D	00000000'	FF	91	00980		CMPB	@CHARPTR, #45	6183	
				17	12	00987		BNEQ	146\$		
		50	00000000'	EF	D0	00989		MOVL	CHARPTR, R0		
		3E	01	A0	91	00990		CMPB	1(R0), #62		
				0A	12	00994		BNEQ	146\$		
	00000000'	EF		02	C0	00996		ADDL2	#2, CHARPTR	6186	
				021F	31	0099D		BRW	174\$	6187	
		1B		51	E9	009A0	146\$:	BLBC	R1, 147\$	6189	
		2A	00000000'	FF	91	009A3		CMPB	@CHARPTR, #42	6190	
				12	12	009AA		BNEQ	147\$		
		0E	0C	AE	E9	009AC		BLBC	12(SP), 147\$	6191	
				00000000'	EF	D6	009B0	INCL	CHARPTR	6194	
		50	00000000'	EF	9E	009B6		MOVAB	C_INDIRECT_TOKEN, R0	6195	
					04	009BD		RET			
				00000000'	EF	D6	009BE	147\$:	INCL	CHARPTR	6200
					5A	D4	009C4	CLRL	BEST_TOKEN_FOUND	6201	
		54		01	CE	009C6		MNEGL	#1, T	6206	
				35	11	009C9		BRB	153\$		
		50	00000000'	EF	9E	009CB	148\$:	MOVAB	TABLEBASE, R0	6204	
58		50	00000000'	EF44	C1	009D2		ADDL3	ADDR_EXPR_OPTBL[1], R0, TOKEN		
		50	0C	A8	9A	009DB		MOVZBL	12(TOKEN), R0	6205	
01	00	0D	A8	50	2D	009DF		CMPC5	R0, 13(TOKEN), #0, #1, (STARTPTR)	6206	
				66		009E5					
				18	12	009E6		BNEQ	153\$		
		0C	0C	AE	E9	009E8		BLBC	12(SP), 151\$	6209	
		02		68	91	009EC		CMPB	(TOKEN), #2	6210	
				03	12	009EF		BNEQ	150\$		
				0352	31	009F1	149\$:	BRW	193\$		
		05	0C	AE	E8	009F4	150\$:	BLBS	12(SP), 152\$	6211	
		02		68	91	009F8	151\$:	CMPB	(TOKEN), #2	6212	
				F4	12	009FB		BNEQ	149\$		
		5A		58	D0	009FD	152\$:	MOVL	TOKEN, BEST_TOKEN_FOUND	6217	
C3		54	00000000'	EF	F2	00A00	153\$:	AOBLSS	ADDR_EXPR_OPTBL-4, 1, 148\$	6202	

03C2	00F9	0A	00000000*	EF	00	00000000G	00	03C2	0019	03C2	03C2	01BC	5A	D5	00A08	TSTL	BEST_TOKEN_FOUND	6223
		50			5A	00	00A0A	1548:	BEQL	1558			04	13	00A0A	BEQL		
					5A	00	00A0C		MOVL	BEST_TOKEN_FOUND, R0			04	04	00A0F	RET		
					56	00	00A10	1558:	MOVL	STARTPTR, CHARPTR			04	04	00A10	MOVL		6224
					00	8F	00A17	1568:	CASEB	DBG\$GB_LANGUAGE, #0, #10			04	8F	00A17	CASEB		6233
					03C2		00A1F	1578:	.WORD	2038-1578,-			03C2		00A1F			
					03C2		00A27			1588-1578,-			03C2		00A27			
					0267		00A2F			2038-1578,-			01BC		00A2F			
										2038-1578,-								
										2038-1578,-								
										2038-1578,-								
										2038-1578,-								
										2038-1578,-								
										2038-1578,-								
										1678-1578,-								
										1778-1578,-								
										1868-1578,-								
										2038-1578								
										2038								
										ADDRESS_EXPRESSION, 1598								6248
										12(SP), -1598								
										@CHARPTR, #46								6249
										1598								
										CHARPTR								6252
										FORTAN_INDIRECT_TOKEN, R0								6253
										CHARPTR, R0								6260
										R0								
										(R0), R0								
										CHAR[BL+1[R0], 1608								
										CHARPTR								6261
										1598								
										#2, CHARPTR								6263
										STARTPTR, CHARPTR, R5								6268
										#1, I								
										1628								
										TABLEBASE, R0								6266
										FORTAN_SPECIAL_OPTBL[I], R0, TOKEN								
										12(TOKEN), R0								6267
										R0, 13(TOKEN), #0, R5, (STARTPTR)								
										1628								
										1938								
										FORTAN_SPECIAL_OPTBL-4, I, 1618								6264
										PRIDTBL, R0								6277
										-4(R0), 8(SP)								
										#1, I								
										1648								
										TABLEBASE, R0								6279
										@PRIDTBL[I], R0, PRID								
										8(PRID), R1								6280
										STARTPTR, CHARPTR, R0								6281
										R1, 9(PRID), #0, R0, (STARTPTR)								6280
										1648								
										8(PRID), R0								6284
										R0, 9(PRID), TOKENBUFFER+1								
										8(PRID), TOKENBUFFER								6285

C4	SA	08	02DF	31	00AF4	BRW	2018	6286
	03	0C	AE	F2	00AF7	AOBLSS	8(SP), 1, 1638	6277
			AE	F9	00AFC	BLBC	12(SP), 1668	6294
	2E		02DE	31	00B00	BRW	2038	
			66	91	00B03	CMPB	(STARTPTR), #46	
			F8	12	00B06	BNEQ	1658	
00000000'	EF	01	A6	9E	00B08	MOVAB	1(R6), CHARPTR	6297
	50	00000000'	EF	9E	00B10	MOVAB	FORTTRAN_DOT_TOKEN, R0	6298
				04	00B17	RET		
	26	00000000'	FF	91	00B18	CMPB	@CHARPTR, #38	6316
			31	12	00B1F	BNEQ	1708	
		00000000'	EF	D6	00B21	INCL	CHARPTR	6319
	26	00000000'	FF	91	00B27	CMPB	@CHARPTR, #38	6320
			0E	12	00B2E	BNEQ	1688	
		00000000'	EF	D6	00B30	INCL	CHARPTR	6323
	50	00000000'	EF	9E	00B36	MOVAB	C_AND_TOKEN, R0	6324
				04	00B3D	RET		
	08	0C	AE	E9	00B3E	BLBC	12(SP), 1698	6327
	50	00000000'	EF	9E	00B42	MOVAB	C_ADDR_OF_TOKEN, R0	
				04	00B49	RET		
	50	00000000'	EF	9E	00B4A	MOVAB	C_BIT_AND_TOKEN, R0	6328
				04	00B51	RET		
	2B	00000000'	FF	91	00B52	CMPB	@CHARPTR, #43	6335
			2A	12	00B59	BNEQ	1728	
		00000000'	EF	D6	00B5B	INCL	CHARPTR	6338
	2B	00000000'	FF	91	00B61	CMPB	@CHARPTR, #43	6339
			13	12	00B68	BNEQ	1718	
		00000000'	EF	D6	00B6A	INCL	CHARPTR	6342
		00028F78	8F	DD	00B70	PUSHL	#167800	6357
00000000G	00		01	FB	00B76	CALLS	#1, LIBSSIGNAL	
	50	00000000'	EF	9E	00B7D	MOVAB	C_ADD_TOKEN, R0	6361
				04	00B84	RET		
	2D	00000000'	FF	91	00B85	CMPB	@CHARPTR, #45	6368
			75	12	00B8C	BNEQ	1798	
		00000000'	EF	D6	00B8E	INCL	CHARPTR	6371
	2D	00000000'	FF	91	00B94	CMPB	@CHARPTR, #45	6372
			13	12	00B9B	BNEQ	1738	
		00000000'	EF	D6	00B9D	INCL	CHARPTR	6375
		00028F78	8F	DD	00BA3	PUSHL	#167800	6389
00000000G	00		01	FB	00BA9	CALLS	#1, LIBSSIGNAL	
	3E	00000000'	FF	91	00BB0	CMPB	@CHARPTR, #62	6393
			0E	12	00BB7	BNEQ	1758	
		00000000'	EF	D6	00BB9	INCL	CHARPTR	6396
	50	00000000'	EF	9E	00BBF	MOVAB	C_ARROW_TOKEN, R0	6397
				04	00BC6	RET		
	08	0C	AE	E9	00BC7	BLBC	12(SP), 1768	6400
	50	00000000'	EF	9E	00BCB	MOVAB	C_MINUS_TOKEN, R0	6402
				04	00BD2	RET		6404
	50	00000000'	EF	9E	00BD3	MOVAB	C_SUB_TOKEN, R0	
				04	00BDA	RET		
	2A	00000000'	FF	91	00BDB	CMPB	@CHARPTR, #42	6414
			1F	12	00BE2	BNEQ	1798	
	03	0C	AE	E8	00BE4	BLBS	12(SP), 1788	6420
			00BD	31	00BE8	BRW	1858	
		00000000'	EF	D6	00BEB	INCL	CHARPTR	6423
	51	00000000'	EF	D0	00BF1	MOVL	CHARPTR, R1	6424
	50		61	9A	00BF8	MOVZBL	(R1), R0	

	03	00000000'	EF40	01DB	57	FE	A1	9E	00C03	1798:	BLBS	CHARTBL[R0], 1808	
	51	00000000'	EF	61	9A	00C0A	1808:	BRW	2038				6435
	50	00000000'	EF40	02	E1	00C1B	1818:	MOVAB	-2(R1), ENDPTR				6438
0E	9E		51	D0	00C1F			MOVL	CHARTBL[R0]				
	57	00000000'	EF40	DF	00C22			MOVZBL	(R1), R0				
26	9E		01	E1	00C29			PUSHAL	CHARTBL[R0]				6441
	51	00000000'	EF	D6	00C2D	1828:		BBC	#2, 2(SP)+, 1828				6442
	50	00000000'	61	9A	00C3A			MOVL	R1, ENDPTR				6445
C9	9E		01	E0	00C44			PUSHAL	CHARTBL[R0]				6446
BE	9E	00000000'	01	E0	00C4F			BBS	#1, 2(SP)+, 1818				
59	EF	00000000'	01	A7	9E	00C53	1838:	PUSHAL	CHARTBL[R0]				6447
	8F	000000FF	56	C3	00C58			BBS	#2, 2(SP)+, 1818				
			59	D1	00C63			MOVAB	1(R7), CHARPTR				6451
			03	14	00C6A			SUBL3	STARTPTR, CHARPTR, TOKENLEN				6456
			F843	31	00C6C			CMPL	TOKENLEN, #255				6457
		00028982	8F	DD	00C6F	1848:		BGTR	1848				
			F833	31	00C75			BRW	608				
	50	00000000'	EF	D6	00C78	1858:		PUSHL	#166274				
		00000000'	EF	9E	00C7E			BRW	598				
	27	00000000'	FF	91	00C86	1868:		INCL	CHARPTR				6468
			03	13	00C8D			MOVAB	RPG_MULTIPLY_TOKEN, R0				6469
			014F	31	00C8F			RET					
	03	0C	AE	E9	00C92	1878:		CMPB	@CHARPTR, #39				6485
			0108	31	00C96			BEQL	1878				
			08	DD	00C99	1888:		BRW	2038				
	00	00000000G	01	FB	00C9B			BRW	1988				6494
68	EF	00000000'	0E	28	00CA5			PUSHL	#8				6500
	57		01	D0	00CAD			CALLS	#1, DBGSGET_TEMPME				
	50	00000000'	EF	9E	00CB0	1898:		MOVL	R0, TOKEN				6501
	50	00000000'	EF47	C1	00CB7			MOVAB	#14, ADA TICK_TOKEN, (TOKEN)				6505
5B	51		68	9A	00CC0			MOVL	#1, INDEX				6508
	50		68	9A	00CC3			MOVAB	TABLEBASE, R0				
	54	00000000'	EF	D0	00CC6			ADDL3	ADA TICK_TABLE[INDEX], R0, NAMEPTR				6510
50	00	01	AB	51	2D	00CCD		MOVZBL	(NAMEPTR), R1				
				51	2D	00CCD		MOVZBL	(NAMEPTR), R0				
				A4	12	00CD5		MOVL	CHARPTR, R4				
				73	12	00CD5		CMPC5	R1, 1(NAMEPTR), #0, R0, 1(R4)				
	50		68	9A	00CD7			BNEQ	1948				6513
	EF	00000000'	01	A440	9E	00CDA		MOVZBL	(NAMEPTR), R0				
	06		57	80	00CE3			MOVAB	1(R4)[R0], CHARPTR				6514
	50		68	9A	00CE7			MOVW	INDEX, 6(TOKEN)				6515
			50	D6	00CEA			MOVZBL	(NAMEPTR), R0				
	OC	AB	50	90	00CEC			INCL	R0				
	OC	AE	68	9A	00CF0			MOVB	R0, 12(TOKEN)				6516
		SA	50	D0	00CF4			MOVZBL	(NAMEPTR), 12(SP)				6517
SA	20	00000000'	EF	01	2C	00CFB		MOVL	R0, R10				
				69	00D04			MOVAB	13(R8), R9				
				OC	18	00D05		MOVCS	#1, P.AXF, #32, R10, (R9)				
								BGEQ	1908				

SA	20	01	AB	OC	59	D6	00D07	INCL	R9		
					5A	D7	00D09	DECL	R10		
					AE	2C	00D0B	MOVCS	12(SP), 1(NAMEPTR), #32, R10, (R9)		
			50	00000000'	69	9A	00D12				
			08	00000000'	FF	40	00D13	1908:	MOVZBL	@CHARPTR, R0	6527
			9E	00000000'	01	DF	00D1A	PUSHAL	CHARTBL+2(R0)		
				00000000'	EF	E1	00D21	BBC	#1, @ (SP)+, 1918		
			28	00000000'	E6	D6	00D25	INCL	CHARPTR		6528
				00000000'	FF	11	00D2B	BRB	1908		
			01	AB	0C	91	00D2D	1918:	CMPB	@CHARPTR, #40	6530
				00000000'	08	12	00D34	BNEQ	1928		
				00000000'	EF	88	00D36	BISB2	#8, 1(TOKEN)		6533
			01	AB	04	D6	00D3A	INCL	CHARPTR		6534
				00000000'	08	11	00D40	BRB	1938		6530
			01	AB	08	8A	00D42	1928:	BICB2	#8, 1(TOKEN)	6537
			50		58	D0	00D46	1938:	MOVL	TOKEN, R0	6539
					04	04	00D49	RET			
			01		09	F1	00D4A	1948:	ACBL	#9, #1, INDEX, 1898	6505
			14	AE	8F	B0	00D50	MOVW	#985, TOKENBUFFER		6546
				00000000'	EF	D6	00D56	1958:	INCL	CHARPTR	6548
			51	00000000'	FF	9A	00D5C	MOVZBL	@CHARPTR, R1		6549
			50	00000000'	EF	41	00D63	MOVAL	CHARTBL[R1], R0		
			08		60	E8	00D6B	BLBS	(R0), 1968		
			04		01	E0	00D6E	BBS	#1, (R0), 1968		6550
			19		02	E1	00D72	BBC	#2, (R0), 1978		6551
					51	91	00D76	1968:	CMPB	R1, #13	6553
					14	13	00D79	BEQL	1978		
			20		AE	91	00D7B	CMPB	TOKENBUFFER, #32		
					0E	1E	00D7F	BGEQU	1978		
					AE	96	00D81	INCB	TOKENBUFFER		6555
			50		AE	9A	00D84	MOVZBL	TOKENBUFFER, R0		6556
			14	AE40	51	90	00D88	MOVB	R1, TOKENBUFFER[R0]		
					C7	11	00D8D	BRB	1958		6557
					AE	9F	00D8F	1978:	PUSHAB	TOKENBUFFER	6560
					01	DD	00D92	PUSHL	#1		
				00028D30	8F	DD	00D94	PUSHL	#167216		
			00000000G	00	03	FB	00D9A	CALLS	#3, LIBSSIGNAL		
				00000000'	EF	D0	00DA1	1988:	MOVL	CHARPTR, R0	6567
			0D		A0	91	00DA8	CMPB	1(R0), #13		
					06	13	00DAC	BEQL	1998		
			27		A0	91	00DAE	CMPB	2(R0), #39		
					0D	13	00DB2	BEQL	2008		
				00028992	8F	DD	00DB4	1998:	PUSHL	#166290	6569
			00000000G	0C	01	FB	00DBA	CALLS	#1, LIBSSIGNAL		
			14	AE	03	90	00DC1	2008:	MOVB	#3, TOKENBUFFER	6571
				00000000'	FF	F0	00DC5	INSV	@CHARPTR, #0, #24, TOKENBUFFER+1		6572
					03	C0	00DCF	ADDL2	#3, CHARPTR		6573
					AE	9F	00DD6	2018:	PUSHAB	TOKENBUFFER	6574
					01	DD	00DD9	PUSHL	#1		
			0000V	CF	02	FB	00DDB	2028:	CALLS	#2, CREATE_OPERAND_TOKEN	
					04	04	00DE0	RET			
				00000000'	56	D0	00DE1	2038:	MOVL	STARTPTR, CHARPTR	6591
					AE	94	00DE8	CLRB	TOKENBUFFER		6592
					50	D4	00DEB	CLRL	I		6595
				0D	00000000'	FF	40	2048:	CMPB	@CHARPTR[I], #13	
					11	13	00DF5	BEQL	2058		
			15	AE40	00000000'	FF	40	MOVW	@CHARPTR[I], TOKENBUFFER+1[I]		6596

DBGPARSER
V04-000

M 14
16-Sep-1984 02:10:13
14-Sep-1984 12:17:30

VAX-11 B11gs-32 V4.0-742
[DEBUG.SRC]DBGPARSER.B32;1

Page 217
(23)

E5	50	14	AE 96 00E01	INCB	TOKENBUFFER	: 6597
		14	F3 00E04	AOBLEQ	#20, I, 2048	: 6593
		14	AE 9F 00E08	PUSHAB	TOKENBUFFER	: 6600
			01 DD 00E0B	PUSHL	#1	
	000289E2		8F DD 00E0D	PUSHL	#166370	
00000000G 00			03 FB 00E13	CALLS	#3, LIBSSIGNAL	
			50 D4 00E1A	CLRL	R0	: 6601
			04 00E1C	RET		: 6603

; Routine Size: 3613 bytes, Routine Base: DBG\$CODE + 0A9E

```
6494 6604 1 GLOBAL ROUTINE DBG$PARSER_SET_LANGUAGE(LANGUAGE): NOVALUE =
6495 6605 1
6496 6606 1 FUNCTION
6497 6607 1     This routine sets up the parse tables used by the Parser and Lexical
6498 6608 1     Scanner for a specified language. It is called during the processing
6499 6609 1     of the SET LANGUAGE command.
6500 6610 1
6501 6611 1     Specifically, this routine sets up the Character Table CHARTBL to
6502 6612 1     have the values appropriate for the specified language. It does so
6503 6613 1     by initializing CHRTBL to have the values appropriate for language
6504 6614 1     UNKNOWN and then changing selected character characteristics as spec-
6505 6615 1     ified in the Character Exception Table for the specified language.
6506 6616 1     It also sets up pointers to the Identifier Operator Table, the Opera-
6507 6617 1     tor Character Operator Table, the Number Scanner State Table, the
6508 6618 1     Primary Parser State Table, and the Subscript Terminator Table for
6509 6619 1     the specified language. It calls the routine DBG$EVALOP_SET_LANGUAGE
6510 6620 1     to set up the Operator Information Tables.
6511 6621 1
6512 6622 1 INPUTS
6513 6623 1     LANGUAGE - The language code for the language being SET.
6514 6624 1
6515 6625 1 OUTPUTS
6516 6626 1     NONE
6517 6627 1
6518 6628 1 BEGIN
6519 6629 1
6520 6630 1 LOCAL
6521 6631 1     CEPTR: REF CE_ENTRY,           | Pointer to Character Exception Table
6522 6632 1                                     | entry for one character
6523 6633 1     CETBL: REF VECTOR[.LONG],     | Pointer to Character Exception Table
6524 6634 1                                     | for the language being SET
6525 6635 1     PTR: REF VECTOR[.LONG];       | Pointer to table of language table
6526 6636 1                                     | pointers for language being SET
6527 6637 1
6528 6638 1
6529 6639 1
6530 6640 1
6531 6641 1     | Set PTR to point to the table of pointers for this language which point
6532 6642 1     | to the various language-specific tables. Use language UNKNOWN if we do
6533 6643 1     | not recognize the language code.
6534 6644 1
6535 6645 1     IF (.LANGUAGE GEO DBG$K_MIN_LANGUAGE) AND
6536 6646 1         (.LANGUAGE LEQ DBG$K_MAX_LANGUAGE)
6537 6647 1     THEN
6538 6648 1         PTR = .LANGUAGE_TABLE_PTRS[.LANGUAGE] + TABLEBASE
6539 6649 1
6540 6650 1     ELSE
6541 6651 1         PTR = .LANGUAGE_TABLE_PTRS[DBG$K_UNKNOWN] + TABLEBASE;
6542 6652 1
6543 6653 1
6544 6654 1     | Set up the Character Table CHARTBL for this language. Also make the
6545 6655 1     | global pointer DBG$GL_CHARTBL point to the character table--this pointer
6546 6656 1     | is used in DBG$SOURCE by the SEARCH command.
6547 6657 1
6548 6658 1     DBG$GL_CHARTBL = CHARTBL;
6549 6659 1     CETBL = PTR[0] + TABLEBASE;
6550 6660 1     CH$MOVE(256*ZUPVAL, BASE_CHARACTER_TABLE, CHARTBL);
```

```
6551 6661 2 INCR I FROM 0 TO .CETBL[-1] - 1 DO
6552 6662 BEGIN
6553 6663 CEPTR = CETBL[I];
6554 6664 CHARTBL[CEPTR[CE_CHAR], CHRTBLSL_WHOLE_ENTRY] = .CEPTR[CE_BITS];
6555 6665 END;
6556 6666
6557 6667
6558 6668 ! Set up the various parse table pointers to the tables for this language.
6559 6669
6560 6670 IDENT OPERATOR TABLE = .PTR[1] + TABLEBASE;
6561 6671 OPCHAR OPERATOR TABLE = .PTR[2] + TABLEBASE;
6562 6672 STATE TABLE = .PTR[3] + TABLEBASE;
6563 6673 PRIMARY TABLE = .PTR[4] + TABLEBASE;
6564 6674 SUBSCRIPT_TERM_TBL = .PTR[5] + TABLEBASE;
6565 6675 PRIDTBL = .PTR[6] + TABLEBASE;
6566 6676 BIF TABLE = .PTR[7] + TABLEBASE;
6567 6677 MULTIPLE SUBSCR = .PTR[8];
6568 6678 ENFORCE RECORD = .PTR[9];
6569 6679 CASING SIGNIFICANT = .PTR[10];
6570 6680 COMPONENTS_IN_PATHNAME = .PTR[11];
6571 6681 INCOMPLETE_QUAL = .PTR[12];
6572 6682
6573 6683 ! Initialize the Operator Evaluation tables and the Print tables for
6574 6684 ! the current language.
6575 6685
6576 6686 DBG$EVALOP SET LANGUAGE (.LANGUAGE);
6577 6687 DBG$PRINT_SET_LANGUAGE (.LANGUAGE);
6578 6688 RETURN;
6579 6689
6580 6690 1 END;
```

		07FC 00000					
		5A	00000000	EF	9E	00002	
		59	00000000	EF	9E	00009	
		58	04	AC	D0	00010	
				11	19	00014	
		0A		58	D1	00016	
				0C	14	00019	
		50		6A	9E	0001B	
56		50	2F91	CA	48	C1	0001E
				09	11	00025	
		50		6A	9E	00027	1\$:
56		50	2FB9	CA	C1	0002A	
	00000000	EF		69	9E	00030	2\$:
		50		6A	9E	00037	
57		50		66	C1	0003A	
69	0591	CA	0400	8F	28	0003E	
		51		01	CE	00046	
				0E	11	00049	
		52		67	41	DE	0004B
		50	03	A2	9A	0004F	3\$:
6940	62	18		00	EF	00053	
				.ENTRY		DBG\$PARSER SET LANGUAGE, Save R2,R3,R4,R5,-	
				MOVAB		R6,R7,R8,R9,R10	
				MOVAB		TABLEBASE, R10	
				MOVL		CHARTBL, R9	
				BLSS		LANGUAGE, R8	
				CMPL		1\$	
				BGTR		R8, #10	
				MOVAB		1\$	
				ADDL3		TABLEBASE, R0	
				BRB		LANGUAGE_TABLE_PTRS[R8], R0, PTR	
				MOVAB		2\$	
				ADDL3		TABLEBASE, R0	
				MOVAB		LANGUAGE_TABLE_PTRS+40, R0, PTR	
				MOVAB		CHARTBL, DBG\$GC_CHARTBL	
				ADDL3		TABLEBASE, R0	
				MOVC3		(PTR), R0, CETBL	
				MNEGL		#1024, BASE_CHARACTER_TABLE, CHARTBL	
				BRB		#1, 1	
				MOVAL		4\$	
				MOVZBL		(CETBL)[1], CEPTR	
				EXTZV		3(CEPTR), R0	
						#0, #24, (CEPTR), CHARTBL[R0]	

ED	51	FC	A7	F2	00059	48:	AOBLSS	-4(CETBL), 1, 3\$	6661
	50		6A	9E	0005E		MOVAB	TABLEBASE, R0	6670
040C	C9	04	B640	9E	00061		MOVAB	24(PTR)[R0], IDENT_OPERATOR_TABLE	
	50		6A	9E	00068		MOVAB	TABLEBASE, R0	6671
0418	C9	08	B640	9E	0006B		MOVAB	28(PTR)[R0], OPCHAR_OPERATOR_TABLE	
	50		6A	9E	00072		MOVAB	TABLEBASE, R0	6672
0424	C9	0C	B640	9E	00075		MOVAB	312(PTR)[R0], STATE_TABLE	
	50		6A	9E	0007C		MOVAB	TABLEBASE, R0	6673
041C	C9	10	B640	9E	0007F		MOVAB	316(PTR)[R0], PRIMARY_TABLE	
	50		6A	9E	00086		MOVAB	TABLEBASE, R0	6674
0428	C9	14	B640	9E	00089		MOVAB	320(PTR)[R0], SUBSCRIPT_TERM_TBL	
	50		6A	9E	00090		MOVAB	TABLEBASE, R0	6675
042C	C9	18	B640	9E	00093		MOVAB	324(PTR)[R0], PRIDTBL	
	50		6A	9E	0009A		MOVAB	TABLEBASE, R0	6676
F4	A9	1C	B640	9E	0009D		MOVAB	328(PTR)[R0], BIF_TABLE	
0414	C9	20	A6	D0	000A3		MOVL	32(PTR), MULTIPLE-SUBSCR	6677
0404	C9	24	A6	D0	000A9		MOVL	36(PTR), ENFORCE_RECORD	6678
F8	A9	28	A6	D0	000AF		MOVL	40(PTR), CASING_SIGNIFICANT	6679
0400	C9	2C	A6	D0	000B4		MOVL	44(PTR), COMPONENTS_IN_PATHNAME	6680
0410	C9	30	A6	D0	000BA		MOVL	48(PTR), INCOMPLETE_QUAL	6681
			58	DD	000C0		PUSHL	R8	6686
00000000G	00		01	FB	000C2		CALLS	#1, DBG\$EVALOP_SET_LANGUAGE	
			58	DD	000C9		PUSHL	R8	6687
00000000G	00		01	FB	000CB		CALLS	#1, DBG\$PRINT_SET_LANGUAGE	
			04	00	000D2		RET		6690

; Routine Size: 211 bytes, Routine Base: DBG\$CODE + 18BB

```
6582 6691 1 GLOBAL ROUTINE DBGPSPRIMARY_PARSER(OPERAND_EXPECTED_FLAG, ADDRESS_EXPRESSION,  
6583 6692 1 TERM_LIST, PAREN_NESTING, RET_TOKEN, RET_OPERAND_FLAG): NOVALUE =  
6584 6693 1  
6585 6694 1 FUNCTION  
6586 6695 1 This routine parses Primary Symbols and serves as a get-token routine  
6587 6696 1 for the Expression Parser. It calls the Lexical Scanner to get lexical  
6588 6697 1 tokens from the command line being parsed. It then intercepts tokens  
6589 6698 1 which are part of a Primary Symbol and uses those to build a Primary  
6590 6699 1 Descriptor for the symbol. Lexical tokens which are not part of Prim-  
6591 6700 1 ary Symbols are simply passed through to the Expression Parser. The  
6592 6701 1 result is that the Expression Parser sees a stream of operators and  
6593 6702 1 operands where each Primary Symbol or constant has been preparsed and  
6594 6703 1 packaged as a single operand by the Primary Parser.  
6595 6704 1  
6596 6705 1 A 'Primary Symbol' is defined to be a variable name which may include  
6597 6706 1 pathname qualification, subscripting, data component selection, and  
6598 6707 1 dereferencing. Exactly which of these are allowed depends on the cur-  
6599 6708 1 rent language. Thus 'X' and 'MOD\ROUT\Z' are Primary Symbols and so is  
6600 6709 1 'M\RX(2,3).Y^Z(4)'. In effect, a Primary Symbol is anything that can  
6601 6710 1 be described by a Primary Descriptor (see DBGLIB.REQ).  
6602 6711 1  
6603 6712 1 The Primary Parser emulates a Finite-State Machine (FSM) to parse the  
6604 6713 1 Primary Symbols accepted in the current language. The FMS for the  
6605 6714 1 current language is defined by a Primary Parser State Table which  
6606 6715 1 defines which operators (such as '\', '^', and subscripting) may appear  
6607 6716 1 in which order in a Primary Symbol. For each transition in the FMS,  
6608 6717 1 a semantic routine is executed which builds up the Primary Descriptor  
6609 6718 1 for the current Primary Symbol (or a Value Descriptor if the current  
6610 6719 1 symbol is a constant). The symbol is 'accepted' by the parser if a  
6611 6720 1 transition is reached which returns the completed Primary Descriptor  
6612 6721 1 to the caller. If the symbol is not accepted by the FSM, a syntax  
6613 6722 1 error is signalled.  
6614 6723 1  
6615 6724 1 The Primary Parser is called by the Expression Parser. However, the  
6616 6725 1 Primary Parser will itself call the Expression Parser to pick up sub-  
6617 6726 1 script expressions within Primary Symbols. Hence these two routines  
6618 6727 1 call each other recursively, and their data structures have been set  
6619 6728 1 up so that this recursion will work properly.  
6620 6729 1  
6621 6730 1 INPUTS  
6622 6731 1 OPERAND_EXPECTED_FLAG - A flag which is set to TRUE if the caller  
6623 6732 1 expects to see an operand next. This flag is used to  
6624 6733 1 determine whether certain operators (such as '+') are  
6625 6734 1 prefix (if an operand is expected) or infix (if an  
6626 6735 1 operand is not expected) operators at the current point  
6627 6736 1 in the parsing of an expression.  
6628 6737 1  
6629 6738 1 ADDRESS_EXPRESSION - A flag set to TRUE if we are parsing a DEBUG  
6630 6739 1 Address Expression instead of a language expression. This  
6631 6740 1 affects the parsing of Address Expression operators such  
6632 6741 1 as '+', '-', '*', '/', '^', and '@' which are recognized by  
6633 6742 1 DEBUG rules, not language rules, in Address Expressions.  
6634 6743 1  
6635 6744 1 TERM_LIST - A vector of pointers to Terminator Lexical Token Entries  
6636 6745 1 for the Terminator Tokens which can terminate the expression  
6637 6746 1 being parsed. The vector must be in PLIT form (TERM_LIST[-1]  
6638 6747 1 gives the number of entries) and each pointer is expected to
```

```

6639 6748 1
6640 6749 1
6641 6750 1
6642 6751 1
6643 6752 1
6644 6753 1
6645 6754 1
6646 6755 1
6647 6756 1
6648 6757 1
6649 6758 1
6650 6759 1
6651 6760 1
6652 6761 1
6653 6762 1
6654 6763 1
6655 6764 1
6656 6765 1
6657 6766 1
6658 6767 1
6659 6768 1
6660 6769 1
6661 6770 1
6662 6771 1
6663 6772 1
6664 6773 1
6665 6774 1
6666 6775 1
6667 6776 1
6668 6777 1
6669 6778 1
6670 6779 1
6671 6780 1
6672 6781 1
6673 6782 1
6674 6783 2
6675 6784 2
6676 6785 2
6677 6786 2
6678 6787 2
6679 6788 2
6680 6789 2
6681 6790 2
6682 6791 2
6683 6792 2
6684 6793 2
6685 6794 2
6686 6795 2
6687 6796 2
6688 6797 2
6689 6798 2
6690 6799 2
6691 6800 2
6692 6801 2
6693 6802 2
6694 6803 2
6695 6804 2

```

be relative to TABLEBASE. If there are no terminator tokens other than carriage return, this list is empty (0 entries).

PAREN_NESTING - The current parenthesis nesting depth. This parameter is passed on to the Lexical Scanner which uses it in the detection of expression terminator tokens.

RET_TOKEN - The address of a longword to receive a pointer to an Operator Lexical Token Entry or a Value or Primary Descriptor.

RET_OPERAND_FLAG - The address of a longword to receive a flag saying whether an operator or an operand was returned.

7th parameter - (optional) - if present, indicates partially constructed Primary and contains pointer to descriptor so far. This is used for example, in the C expression (*PTR).COMPONENT, where a Primary is returned from the expression parser for (*PTR) and we call the Primary Parser to pick up the ".COMPONENT".

8th parameter - (optional) - if present, indicated partially constructed Primary and contains starting state.

OUTPUTS

RET_TOKEN - A pointer to an Operator Lexical Token Entry or to a Value or Primary Descriptor is returned to RET_TOKEN. What the returned pointer points to is specified by RET_OPERAND_FLAG.

RET_OPERAND_FLAG - A flag value is returned to RET_OPERAND_FLAG. If a pointer to an Operator Lexical Token Entry was returned to RET_TOKEN, the value FALSE is returned to RET_OPERAND_FLAG. If a pointer to a Primary or Value Descriptor is returned to RET_TOKEN, the value TRUE is returned to RET_OPERAND_FLAG.

BEGIN

MAP

RET_TOKEN: REF VECTOR[1], ! Token pointer return location
RET_OPERAND_FLAG: REF VECTOR[1]; ! Operand returned flag location

BUILTIN

ACTUALCOUNT,
ACTUALPARAMETER;

OWN

TOKEN IS INTEGER: ! Lookup table which states whether a
BITVECTOR[TOKEN\$K_MAX_OPERAND + 1] given operand token is some
PSECT(DBG\$PLIT) form of integer or not
PRESET(
[TOKEN\$K_INTEGER] = TRUE,
[TOKEN\$K_HEX_INTEGER] = TRUE,
[TOKEN\$K_OCT_INTEGER] = TRUE,
[TOKEN\$K_BIN_INTEGER] = TRUE,
[TOKEN\$K_PACK_DECIMAL] = TRUE);

LOCAL


```

6696 6805 2 ACTION,
6697 6806 2 ARG LIST: REF VECTOR [ ,LONG],
6698 6807 2 DUMMY,
6699 6808 2
6700 6809 2 KIND,
6701 6810 2 LAST_OPERAND: REF TOKEN$ENTRY,
6702 6811 2
6703 6812 2 NUMERIC_PATHNAME,
6704 6813 2 OPCODE,
6705 6814 2 OPERAND_EXPECTED,
6706 6815 2
6707 6816 2 PATHDESC: PTH$PATHNAME,
6708 6817 2 PATHSTRING,
6709 6818 2 PATHVECTOR: REF VECTOR[ ,LONG],
6710 6819 2 PLIPTR: REF DBG$PRIMARY,
6711 6820 2 PRID: REF PRID$ENTRY,
6712 6821 2 PRIMPTR: REF DBG$PRIMARY,
6713 6822 2 SAVED_PATHDESC: PTH$PATHNAME,
6714 6823 2 STATE_INDEX,
6715 6824 2
6716 6825 2 STATUS,
6717 6826 2 STRDESC: BLOCK[8,BYTE],
6718 6827 2 SUBSCR_DESC: SUBSCR$DESC,
6719 6828 2 SYMID,
6720 6829 2 TEMPTOKEN: REF TOKEN$ENTRY,
6721 6830 2 TOKEN: REF TOKEN$ENTRY,
6722 6831 2 TYPEID,
6723 6832 2 VALPTR: REF DBG$VALDESC;
6724 6833 2
6725 6834 2
6726 6835 2
6727 6836 2
6728 6837 2
6729 6838 2
6730 6839 2
6731 6840 2
6732 6841 2
6733 6842 2
6734 6843 2
6735 6844 2
6736 6845 2
6737 6846 2
6738 6847 2
6739 6848 2
6740 6849 2
6741 6850 2
6742 6851 2
6743 6852 2
6744 6853 2
6745 6854 2
6746 6855 2
6747 6856 2
6748 6857 2
6749 6858 2
6750 6859 2
6751 6860 2
6752 6861 2

! Current state transition action index
! List of built-in function arguments
! Output param for DEFINE lookup - not
!   used here.
! RST symbol kind for current symbol
! Pointer to the last operand token
!   entry encountered so far
! Flag set if numeric pathname is used
! Operator code for current operator
! Flag set when operand or prefix ope-
!   rator is expected next
! Pathname descriptor
! Pointer to pathname string for messages
! Pointer to pathname vector in PATHDESC
! Pointer value
! Pointer to Predefined Identifier Entry
! Pointer to Primary Descriptor built
! Copy of pathname descriptor
! Current index into the Primary Parser
!   State Table
! Status code returned by RTL routines
! String descriptor for RTL calls
! Holds saved-away subscripts
! SYMID (Symbol ID) for current symbol
! Pointer to token for invocation number
! Pointer to the current Token Entry
! TYPEID (Type ID) for current symbol
! Pointer to a Value Descriptor

! There are two different initialization paths. The normal path is
! when we are picking up a Primary from scratch; that is the ELSE
! clause below.
IF ACTUALCOUNT() GTR 6
THEN
  BEGIN

    ! If we got an Operator Token last time which was not part of the Primary
    ! Symbol we were building, then we saved it in SAVED_TOKEN while we com-
    ! pleted and returned the Primary Descriptor. In that case, return the
    ! input Primary and retain the SAVED_TOKEN value for the next time
    ! Primary Parser is called.
    IF .SAVED_TOKEN NEQ 0
    THEN
      BEGIN
        RET TOKEN[0] = ACTUALPARAMETER(7);
        RETURN;
      END;

    ! This is the case where we call the Primary Parser when we already
    ! have constructed part of the Primary, and we want to pick up the
    ! rest of the Primary. In this case, a pointer to the partially-
```



```
6753 6862      : constructed Primary is passed in as the 7th parameter, and the
6754 6863      : current state in the primary parsing is passed in as the 8th parameter.
6755 6864      : We also initialize LAST_OPERAND to -1 here; it must be non-zero to
6756 6865      : avoid confusing the code below.
6757 6866
6758 6867      LAST_OPERAND = -1;
6759 6868      OPERAND_EXPECTED = .OPERAND_EXPECTED_FLAG;
6760 6869      STATE_INDEX = ACTUALPARAMETER(8);
6761 6870      PRIMPTR = ACTUALPARAMETER(7);
6762 6871      END
6763 6872
6764 6873  ELSE
6765 6874      BEGIN
6766 6875
6767 6876      : If we got an Operator Token last time which was not part of the Primary
6768 6877      : Symbol we were building, then we saved it in SAVED_TOKEN while we com-
6769 6878      : pleted and returned the Primary Descriptor. In that case, return the
6770 6879      : saved token to the Expression Parser now and do no more.
6771 6880
6772 6881      IF .SAVED_TOKEN NEQ 0
6773 6882      THEN
6774 6883          BEGIN
6775 6884              TOKEN = .SAVED_TOKEN;
6776 6885              SAVED_TOKEN = 0;
6777 6886              RET_TOKEN[0] = .TOKEN;
6778 6887              RET_OPERAND_FLAG[0] = FALSE;
6779 6888              RETURN;
6780 6889              END;
6781 6890
6782 6891      : Initialize LAST_OPERAND to be null (no operands have been encountered
6783 6892      : yet). Also initialize the OPERAND_EXPECTED flag and various other local
6784 6893      : variables and data structures.
6785 6894
6786 6895      LAST_OPERAND = 0;
6787 6896      OPERAND_EXPECTED = .OPERAND_EXPECTED_FLAG;
6788 6897      STATE_INDEX = 0;
6789 6898      PRIMPTR = 0;
6790 6899      NUMERIC_PATHNAME = FALSE;
6791 6900      CH$FILLTO, DBG$K_PATHNAME SIZE*%UPVAL, PATHDESC);
6792 6901      PATHVECTOR = PATHDESC[PATH$A_PATHVECTOR];
6793 6902      CH$FILL(0, SUBSCR_DESC_SIZE, SUBSCR_DESC);
6794 6903      PLIPTR = 0;
6795 6904      END;
6796 6905
6797 6906      : Loop through all lexical tokens on the input line being parsed until we
6798 6907      : reach a terminator operator.
6799 6908
6800 6909      WHILE TRUE DO
6801 6910          BEGIN
6802 6911
6803 6912      : Get the next lexical token. DBG$LEXICAL_SCANNER picks up the next
6804 6913      : token using the rules of the currently set language.
6805 6914
6806 6915      TOKEN = DBG$LEXICAL_SCANNER(.OPERAND_EXPECTED,
```

```

6810
6811
6812
6813
6814
6815
6816
6817
6818
6819
6820
6821
6822
6823
6824
6825
6826
6827
6828
6829
6830
6831
6832
6833
6834
6835
6836
6837
6838
6839
6840
6841
6842
6843
6844
6845
6846
6847
6848
6849
6850
6851
6852
6853
6854
6855
6856
6857
6858
6859
6860
6861
6862
6863
6864
6865
6866

```

```

ADDRESS EXPRESSION, TERM LIST, PAREN_NESTING);
IF .DBG$GL_DEVELOPER[2] THEN DUMP_TOKEN(.TOKEN);

! Check for an invocation number. An invocation number consists of an
! integer constant in a pathname where an operator is expected. If
! this is an invocation number, we convert it to the invocation number
! postfix operator which then passes through the rest of the code below
! in the normal way.
IF (.PRIMPTR EQL 0) AND
  (NOT .OPERAND_EXPECTED) AND
  (.LAST_OPERAND NEQ 0) AND
  (.TOKEN[TOKEN$B_KIND] EQL TOKEN$K_OPERATOR) AND
  .TOKEN_IS_INTEGER(.TOKEN[TOKEN$B_CODE])
THEN
  BEGIN
    TEMPTOKEN = DBG$GET_TEMPMEM(TOKEN$K_ENTSIZE_OPERATOR +
      (.TOKEN[TOKEN$B_LENGTH] + %UPVAL)/%UPVAL);
    TEMPTOKEN[TOKEN$B_KIND] = TOKEN$K_POSTFIX_OP;
    TEMPTOKEN[TOKEN$B_PRIMARY] = TRUE;
    TEMPTOKEN[TOKEN$B_CODE] = TOKEN$K_INVOCNUM;
    CH$MOVE(.TOKEN[TOKEN$B_LENGTH] + 1,
      TOKEN[TOKEN$B_LENGTH], TEMPTOKEN[TOKEN$B_OPLEN]);
    TOKEN = .TEMPTOKEN;
  END;

! Handle operands. If this is an operand, check that we are actually
! expecting an operand at this point. Save a pointer to the operand
! and loop to get the next token.
IF .TOKEN[TOKEN$B_KIND] EQL TOKEN$K_OPERATOR
THEN
  BEGIN
    IF NOT .OPERAND_EXPECTED
    THEN
      SIGNAL(DBG$MISINVOPER, 1, TOKEN[TOKEN$B_LENGTH]);

    OPERAND_EXPECTED = FALSE;
    LAST_OPERAND = .TOKEN;
  END

! Handle operators. If this operator is not part of the current Prim-
! ary Symbol, we save it while building and returning a descriptor for
! the Primary Symbol. If the operator is part of the current Primary
! Symbol, we add to the Primary we are building and loop to pick up
! more of the Primary.
ELSE
  BEGIN
    ! If this operator is not part of the Primary Symbol we are build-
    ! ing (if any), then it is a language or address expression opera-
    ! tor. Save it for the next call on DBG$PRIMARY_PARSER and use the

```

6867	6976	4
6868	6977	4
6869	6978	4
6870	6979	4
6871	6980	4
6872	6981	4
6873	6982	4
6874	6983	4
6875	6984	4
6876	6985	4
6877	6986	4
6878	6987	4
6879	6988	4
6880	6989	4
6881	6990	4
6882	6991	4
6883	6992	4
6884	6993	4
6885	6994	4
6886	6995	4
6887	6996	4
6888	6997	4
6889	6998	4
6890	6999	4
6891	7000	4
6892	7001	4
6893	7002	4
6894	7003	4
6895	7004	4
6896	7005	4
6897	7006	4
6898	7007	4
6899	7008	4
6900	7009	4
6901	7010	4
6902	7011	4
6903	7012	4
6904	7013	4
6905	7014	4
6906	7015	4
6907	7016	4
6908	7017	4
6909	7018	4
6910	7019	4
6911	7020	4
6912	7021	4
6913	7022	4
6914	7023	4
6915	7024	4
6916	7025	4
6917	7026	4
6918	7027	4
6919	7028	4
6920	7029	4
6921	7030	4
6922	7031	4
6923	7032	4

Terminator Token to close out the Primary Symbol we are building
(if we are building one).

IF NOT .TOKEN[TOKEN\$B_PRIMARY]
THEN

BEGIN

! If no operand was started, return the operator immediately.

IF .LAST_OPERAND EQL 0
THEN

BEGIN

RET_TOKEN[0] = .TOKEN;
RET_OPERAND_FLAG[0] = FALSE;
RETURN;
END;

! An operand is present--save the language operator and set
things up to close out and return the operand.

SAVED_TOKEN = .TOKEN;
TOKEN = PRIMARY_TERM_TOKEN;
RET_OPERAND_FLAG[0] = TRUE;
END;

! We now have a Primary operator. Check that an operator was
expected unless this is a prefix operator (which is okay when
we expect an operand). This check catches many kinds of ill-
formed Primary Symbols. Also say that we expect an operand
next unless this is a postfix operator.

IF (.OPERAND_EXPECTED AND
T.TOKEN[TOKEN\$B_KIND] NEQ TOKEN\$K_PREFIX_OP) OR
((NOT .OPERAND_EXPECTED) AND
(.TOKEN[TOKEN\$B_KIND] EQL TOKEN\$K_PREFIX_OP))

THEN

SIGNAL(DBG\$MISOPENIS, 1, TOKEN[TOKEN\$B_OPLEN]);

IF .TOKEN[TOKEN\$B_KIND] NEQ TOKEN\$K_PREFIX_OP

THEN

IF .LAST_OPERAND EQL 0

THEN

SIGNAL(DBG\$MISOPENIS, 1, TOKEN[TOKEN\$B_OPLEN])

ELSE

BEGIN

IF .LAST_OPERAND NEQ -1

THEN

BEGIN

IF (.LAST_OPERAND[TOKEN\$B_KIND] NEQ TOKEN\$K_IDENTIFIER)

THEN

SIGNAL(DBG\$MISOPENIS, 1, TOKEN[TOKEN\$B_OPLEN]);

END;

END;

6924 7033 4
6925 7034 4
6926 7035 4
6927 7036 4
6928 7037 4
6929 7038 4
6930 7039 4
6931 7040 4
6932 7041 4
6933 7042 4
6934 7043 4
6935 7044 4
6936 7045 4
6937 7046 4
6938 7047 4
6939 7048 4
6940 7049 4
6941 7050 5
6942 7051 5
6943 7052 5
6944 7053 5
6945 7054 5
6946 7055 5
6947 7056 4
6948 7057 4
6949 7058 4
6950 7059 4
6951 7060 4
6952 7061 4
6953 7062 4
6954 7063 4
6955 7064 4
6956 7065 4
6957 7066 4
6958 7067 4
6959 7068 4
6960 7069 4
6961 7070 4
6962 7071 4
6963 7072 4
6964 7073 4
6965 7074 4
6966 7075 4
6967 7076 4
6968 7077 4
6969 7078 4
6970 7079 4
6971 7080 5
6972 7081 5
6973 7082 5
6974 7083 5
6975 7084 5
6976 7085 5
6977 7086 5
6978 7087 5
6979 7088 4
6980 7089 4

```
IF .TOKEN[TOKEN$B_KIND] NEQ TOKEN$K_POSTFIX_OP
THEN
  OPERAND_EXPECTED = TRUE;
```

```
! Get the Operator Code for this Primary Operator and loop through
! the transitions for the current state in the Primary Parser State
! Table until we find a transition for this operator. If we find
! no such transition (PRIMARY$B_OPCODE field zero), the current
! operator is not allowed in this context, so we signal a syntax
! error. If the transition is allowed, we pick up its action index
! and the next state in the FSM.
```

```
OPCODE = .TOKEN[TOKEN$B_CODE];
WHILE .PRIMARY_TABLE[.STATE_INDEX, PRIMARY$B_OPCODE] NEQ .OPCODE DO
  BEGIN
    IF .PRIMARY_TABLE[.STATE_INDEX, PRIMARY$B_OPCODE] EQL 0
    THEN
      SIGNAL(DBG$_SYNERREXPR, 1, TOKEN[TOKEN$B_OPLEN]);

      STATE_INDEX = .STATE_INDEX + 1;
    END;
```

```
ACTION = .PRIMARY_TABLE[.STATE_INDEX, PRIMARY$B_ACTION];
STATE_INDEX = .PRIMARY_TABLE[.STATE_INDEX, PRIMARY$B_NEXTSTATE];
```

```
! Execute the action routine associated with this state transition.
CASE .ACTION FROM PRIMARY$K_MIN_ACTION TO PRIMARY$K_MAX_ACTION OF
  SET
```

```
! Handle Global Symbol backslash operator (prefix '\'). Do
! nothing at this point.
```

```
[PRIMARY$K_ACT_START_GBL]:
  0;
```

```
! Handle terminator after Global Symbol backslash. Just pick
! up the global symbol name and create a Primary Descriptor
! for it. Then exit from the parse loop.
```

```
[PRIMARY$K_ACT_GBL_TERM]:
  BEGIN
    PATHDESC[PTH$B_TOTCNT] = 1;
    PATHDESC[PTH$B_PATHCNT] = 1;
    PATHVECTOR[0] = UPLIT BYTE(0);
    APPEND TO PATHNAME(PATHDESC, .LAST_OPERAND, NOT_REC_COMP);
    PRIMPTR = "PATHNAME_TO_PRIMARY(PATHDESC, SUBSCR_DESC,
    .PLIPTR, SAVED_PATHDESC);

    EXITLOOP;
  END;
```


6981	7090	4
6982	7091	4
6983	7092	4
6984	7093	4
6985	7094	4
6986	7095	4
6987	7096	4
6988	7097	4
6989	7098	5
6990	7099	5
6991	7100	5
6992	7101	5
6993	7102	5
6994	7103	5
6995	7104	5
6996	7105	5
6997	7106	5
6998	7107	5
6999	7108	6
7000	7109	6
7001	7110	6
7002	7111	6
7003	7112	6
7004	7113	6
7005	7114	6
7006	7115	6
7007	7116	6
7008	7117	6
7009	7118	6
7010	7119	6
7011	7120	6
7012	7121	6
7013	7122	6
7014	7123	6
7015	7124	6
7016	7125	6
7017	7126	6
7018	7127	6
7019	7128	6
7020	7129	6
7021	7130	6
7022	7131	6
7023	7132	5
7024	7133	5
7025	7134	5
7026	7135	4
7027	7136	4
7028	7137	4
7029	7138	4
7030	7139	4
7031	7140	4
7032	7141	4
7033	7142	4
7034	7143	5
7035	7144	5
7036	7145	5
7037	7146	5

```
! Handle backslash immediately after the start of the symbol.
! If the last operand was a number, we have a numeric pathname
! and we put that into the Pathname Descriptor and set the
! numeric pathname flag. If it is an identifier, we simply
! append it to the Pathname Descriptor as is.
```

```
[PRIMARY$K_ACT_START_SLASH]:
BEGIN
```

```
! If the last operand is an integer, then it constitutes
! a numeric pathname. We put that in the Pathname Descrip-
! tor and set the NUMERIC_PATHNAME flag (which means that
! no additional pathname qualification is allowed).
```

```
IF .TOKEN_IS_INTEGER[.LAST_OPERAND[TOKEN$W_CODE]]
THEN
```

```
BEGIN
  STRDESC[DSC$B_DTYPE] = DSC$K_DTYPE_T;
  STRDESC[DSC$B_CLASS] = DSC$K_CLASS_S;
  STRDESC[DSC$W_LENGTH] = .LAST_OPERAND[TOKEN$B_LENGTH];
  STRDESC[DSC$A_POINTER] = .LAST_OPERAND[TOKEN$A_NAME];
  STATUS = OTSS[VT_TI_L(STRDESC, PATHDESC[PTH$L_INVOCNUM]);
  IF NOT .STATUS
  THEN
```

```
    SIGNAL(DBG$ ILLPATHELEM, 1,
           LAST_OPERAND[TOKEN$B_LENGTH], .STATUS);
```

```
    PATHDESC[PTH$B_LOCINVOC] = 1;
    PATHDESC[PTH$B_TOTCNT] = 1;
    PATHDESC[PTH$B_PATHCNT] = 1;
    PATHVECTOR[0] = UPLIT BYTE(0);
    NUMERIC_PATHNAME = TRUE;
  END
```

```
! Otherwise simply append the last operand to the current
! Pathname Descriptor. (The append routine checks that
! the operand is an identifier.)
```

```
ELSE
  APPEND_TO_PATHNAME(PATHDESC, .LAST_OPERAND, NOT_REC_COMP);
```

```
END;
```

```
! Handle an invocation number immediately after the first or a
! later pathname component of the symbol. Check the invocation
! number for validity and fill it into the Pathname Descriptor.
```

```
[PRIMARY$K_ACT_SLASH_INVOCNUM]:
BEGIN
```

```
! Check that an invocation number has not already been
```

```

7038 7147 5
7039 7148 5
7040 7149 5
7041 7150 5
7042 7151 5
7043 7152 5
7044 7153 5
7045 7154 5
7046 7155 5
7047 7156 5
7048 7157 5
7049 7158 5
7050 7159 5
7051 7160 5
7052 7161 5
7053 7162 5
7054 7163 5
7055 7164 5
7056 7165 5
7057 7166 5
7058 7167 5
7059 7168 5
7060 7169 5
7061 7170 5
7062 7171 5
7063 7172 5
7064 7173 5
7065 7174 5
7066 7175 5
7067 7176 5
7068 7177 5
7069 7178 5
7070 7179 5
7071 7180 5
7072 7181 5
7073 7182 5
7074 7183 5
7075 7184 5
7076 7185 6
7077 7186 6
7078 7187 6
7079 7188 7
7080 7189 7
7081 7190 7
7082 7191 7
7083 7192 7
7084 7193 7
7085 7194 7
7086 7195 7
7087 7196 7
7088 7197 7
7089 7198 7
7090 7199 7
7091 7200 8
7092 7201 8
7093 7202 8
7094 7203 8

```

```

: specified--if one has, signal an error.
:
: IF .PATHDESC[PTH$B_LOCINVO] NEQ 0
: THEN
:   SIGNAL(DBG$B_TOOMANINV);

```

```

: Convert the invocation number to internal form and fill
: it into the Pathname Descriptor.
:
: STRDESC[DSC$B_DTYPE] = DSC$K_DTYPE_T;
: STRDESC[DSC$B_CLASS] = DSC$K_CLASS_S;
: STRDESC[DSC$B_LENGTH] = .TOKEN[TOKEN$B_OPLEN];
: STRDESC[DSC$B_POINTER] = .TOKEN[TOKEN$B_OPNAME];
: STATUS = OTSSCVT_T1_L(STRDESC, PATHDESC[PTH$B_INVOCNUM]);
: IF NOT .STATUS
: THEN
:   SIGNAL(DBG$B_ILLPATHELEM, 1, .TOKEN[TOKEN$B_OPLEN],
:     DBG$B_UNACVT, 3, UPLIT BYTE(%ASCII 'decimal '),
:     .STRDESC, UPLIT BYTE(%ASCII 'longword integer'),
:     .STATUS);

```

```

: PATHDESC[PTH$B_LOCINVO] = .PATHDESC[PTH$B_TOTCNT] + 1;
: END;

```

```

: Handle dot immediately after the start of the symbol. Append
: the last operand to the current Pathname Descriptor and build
: the first part of a Primary Descriptor for it.
:
: [PRIMARY$K_ACT_START_DOT]:
: BEGIN
:   LABEL TEMP_BLOCK;

```

```

: If the last operand was an identifier, we append it to
: the current Pathname Descriptor and convert that to a
: Primary Descriptor.
:
: TEMP_BLOCK: BEGIN
: IF .LAST_OPERAND[TOKEN$B_CODE] EQL TOKEN$K_IDENTIFIER
: THEN
:   BEGIN

```

```

: First check for DEFINED symbols.
: Check that no invocation number is present.
:
: IF .PATHDESC[PTH$B_LOCINVO] EQL 0
: THEN

```

```

: Look up the symbol in the DEFINE symbol table.
:
: BEGIN
: IF DBG$DEF_SYM_FIND (LAST_OPERAND [TOKEN$B_LENGTH],
:   KIND, PRIMPTR,
:   DUMMY, DUMMY)

```

7095 7204 8
7096 7205 9
7097 7206 9
7098 7207 9
7099 7208 9
7100 7209 10
7101 7210 10
7102 7211 10
7103 7212 10
7104 7213 10
7105 7214 10
7106 7215 10
7107 7216 10
7108 7217 10
7109 7218 10
7110 7219 9
7111 7220 8
7112 7221 7
7113 7222 6
7114 7223 6
7115 7224 6
7116 7225 6
7117 7226 5
7118 7227 5
7119 7228 4
7120 7229 4
7121 7230 4
7122 7231 4
7123 7232 4
7124 7233 4
7125 7234 4
7126 7235 4
7127 7236 5
7128 7237 5
7129 7238 4
7130 7239 4
7131 7240 4
7132 7241 4
7133 7242 4
7134 7243 4
7135 7244 5
7136 7245 5
7137 7246 4
7138 7247 4
7139 7248 4
7140 7249 4
7141 7250 4
7142 7251 4
7143 7252 4
7144 7253 4
7145 7254 4
7146 7255 5
7147 7256 5
7148 7257 5
7149 7258 5
7150 7259 5
7151 7260 5

```
THEN
  BEGIN
    IF .KIND EQL DEFINE_ADDRESS
    OR .KIND EQL DEFINE_VALUE
    THEN
      BEGIN
        ! We have found a matching DEFINED symbol.
        ! Copy the descriptor into temporary memory.
        ! (Fourth parameter FALSE <-> copy into tempmem).
        DBG$NCPY_DESC (.PRIMPTR, PRIMPTR,
                      DUMMY, FALSE);
        LEAVE TEMP_BLOCK;
      END;
    END;
  END;
  END;
  APPEND_TO_PATHNAME(PATHDESC, .LAST_OPERAND, NOT_REC_COMP);
  PRIMPTR = PATHNAME_TO_PRIMARY(PATHDESC, SUBSCR_DESC,
                                .PLIPTR, SAVED_PATHDESC);
  END; ! TEMP_BLOCK
  END;

! Handle a dot immediately after the start of the symbol.
! In PLI we collect all the record components before
! calling PATHNAME_TO_PRIMARY.
[PRIMARY$K_ACT_START_DOT_PLI]:
  BEGIN
    APPEND_TO_PATHNAME(PATHDESC, .LAST_OPERAND, NOT_REC_COMP);
  END;

! Handle a dot after the start of the symbol in COBOL.
[PRIMARY$K_ACT_START_DOT_COB]:
  BEGIN
    APPEND_TO_PATHNAME(PATHDESC, .LAST_OPERAND, COB_REC_COMP);
  END;

! Handle a subscript parenthesis immediately after the start of
! the symbol. Append the last operand to the Pathname Descrip-
! tor, build a partial Primary Descriptor, and then pick up the
! subscript expressions within the parentheses.
[PRIMARY$K_ACT_START_SUBSCR]:
  BEGIN
    LABEL TEMP_BLOCK;

    ! If the last operand was an identifier, we append it to
    ! the current Pathname Descriptor and convert that to a
    ! Primary Descriptor.
```

7152 7261 5
7153 7262 6
7154 7263 6
7155 7264 6
7156 7265 7
7157 7266 7
7158 7267 7
7159 7268 7
7160 7269 7
7161 7270 7
7162 7271 7
7163 7272 7
7164 7273 7
7165 7274 7
7166 7275 7
7167 7276 7
7168 7277 8
7169 7278 8
7170 7279 8
7171 7280 8
7172 7281 8
7173 7282 9
7174 7283 9
7175 7284 9
7176 7285 9
7177 7286 10
7178 7287 10
7179 7288 10
7180 7289 10
7181 7290 10
7182 7291 10
7183 7292 10
7184 7293 10
7185 7294 10
7186 7295 10
7187 7296 9
7188 7297 8
7189 7298 7
7190 7299 6
7191 7300 6
7192 7301 6
7193 7302 6
7194 7303 5
7195 7304 5
7196 7305 5
7197 7306 4
7198 7307 4
7199 7308 4
7200 7309 4
7201 7310 4
7202 7311 4
7203 7312 4
7204 7313 4
7205 7314 5
7206 7315 5
7207 7316 5
7208 7317 4

```

TEMP_BLOCK: BEGIN
IF .LAST_OPERAND[TOKEN$W_CODE] EQL TOKEN$K_IDENTIFIER
THEN
    BEGIN

        ! First check for DEFINED symbols.
        ! Check that no invocation number is present.
        IF .PATHDESC[PTH$B_LOCINVOC] EQL 0
        THEN

            ! Look up the symbol in the DEFINE symbol table.
            BEGIN
            IF DBG$DEF_SYM_FIND (LAST_OPERAND [TOKEN$B_LENGTH],
                                KIND, PRIMPTR,
                                DUMMY, DUMMY)
            THEN
                BEGIN
                IF .KIND EQL DEFINE_ADDRESS
                OR .KIND EQL DEFINE_VALUE
                THEN
                    BEGIN

                        ! We have found a matching DEFINED symbol.
                        ! Copy the descriptor into temporary memory.
                        ! (Fourth parameter FALSE <-> copy into tempmem).
                        DBG$NCOPY_DESC (.PRIMPTR, PRIMPTR,
                                      DUMMY, FALSE);
                        LEAVE TEMP_BLOCK;
                    END;
                END;
            END;
        END;
        APPEND TO PATHNAME(PATHDESC, .LAST_OPERAND, NOT_REC_COMP);
        PRIMPTR = PATHNAME_TO_PRIMARY(PATHDESC, SUBSCR_DESC,
                                      .PLIPTR, SAVED_PATHDESC, TRUE);
        END; ! TEMP_BLOCK

        GET_SUBSCRIPTS(.PRIMPTR);
        END;

        ! Handle a subscript immediately after the start of the
        ! symbol. In PLI we save away the subscripts and do not
        ! build them into the Primary until later.
        [PRIMARY$K_ACT_START_SUBSCR_PLI]:
        BEGIN
        APPEND TO PATHNAME(PATHDESC, .LAST_OPERAND, NOT_REC_COMP);
        SAVE_SUBSCRIPTS(PATHDESC, SUBSCR_DESC);
        END;
    
```



```

7209 7318 4
7210 7319 4
7211 7320 4
7212 7321 4
7213 7322 4
7214 7323 4
7215 7324 4
7216 7325 4
7217 7326 4
7218 7327 4
7219 7328 5
7220 7329 5
7221 7330 5
7222 7331 5
7223 7332 6
7224 7333 6
7225 7334 6
7226 7335 7
7227 7336 7
7228 7337 7
7229 7338 7
7230 7339 7
7231 7340 7
7232 7341 7
7233 7342 7
7234 7343 7
7235 7344 7
7236 7345 7
7237 7346 7
7238 7347 8
7239 7348 8
7240 7349 8
7241 7350 8
7242 7351 8
7243 7352 9
7244 7353 9
7245 7354 9
7246 7355 9
7247 7356 10
7248 7357 10
7249 7358 10
7250 7359 10
7251 7360 10
7252 7361 10
7253 7362 10
7254 7363 10
7255 7364 10
7256 7365 10
7257 7366 9
7258 7367 8
7259 7368 7
7260 7369 7
7261 7370 7
7262 7371 6
7263 7372 6
7264 7373 6
7265 7374 6

```

```

! Handle a subscript parenthesis immediately after the start of
! the symbol. Append the last operand to the Pathname Descrip-
! tor, build a partial Primary Descriptor, and then pick up the
! subscript expressions within the square brackets. BLISS needs to
! be special cased because BLISS structures are represented
! differently in the RST and DST.

```

```

[PRIMARY$K_ACT_START_SUBSCR_BLI]:

```

```

BEGIN

```

```

LABEL TEMP_BLOCK;

```

```

! If the last operand was an identifier, we

```

```

TEMP_BLOCK: BEGIN

```

```

IF .LAST_OPERAND[TOKEN$W_CODE] EQL TOKEN$K_IDENTIFIER

```

```

THEN

```

```

BEGIN

```

```

! First check for DEFINED symbols.

```

```

! Check that no invocation number is present.

```

```

IF .PATHDESC[PTH$B_LOCINVOC] EQL 0

```

```

THEN

```

```

! Look up the symbol in the DEFINE symbol table.

```

```

BEGIN

```

```

IF DBG$DEF_SYM_FIND (LAST_OPERAND [TOKEN$B_LENGTH],
                     KIND, PRIMPTR,
                     DUMMY, DUMMY)

```

```

THEN

```

```

BEGIN

```

```

IF .KIND EQL DEFINE_ADDRESS

```

```

OR .KIND EQL DEFINE_VALUE

```

```

THEN

```

```

BEGIN

```

```

! We have found a matching DEFINED symbol.

```

```

! Copy the descriptor into temporary memory.

```

```

! (Fourth parameter FALSE <-> copy into tempmem).

```

```

DBG$NCOPY_DESC (.PRIMPTR, PRIMPTR,
                DUMMY, FALSE);

```

```

LEAVE TEMP_BLOCK;

```

```

END;

```

```

END;

```

```

END;

```

```

END

```

```

ELSE

```

```

! If the last operand was not an identifier then
! it is not legal to select a component.

```

7266	7375	6
7267	7376	6
7268	7377	6
7269	7378	6
7270	7379	6
7271	7380	6
7272	7381	6
7273	7382	6
7274	7383	6
7275	7384	6
7276	7385	5
7277	7386	5
7278	7387	5
7279	7388	4
7280	7389	4
7281	7390	4
7282	7391	4
7283	7392	4
7284	7393	4
7285	7394	4
7286	7395	4
7287	7396	4
7288	7397	4
7289	7398	4
7290	7399	5
7291	7400	5
7292	7401	5
7293	7402	5
7294	7403	5
7295	7404	5
7296	7405	5
7297	7406	6
7298	7407	6
7299	7408	6
7300	7409	7
7301	7410	7
7302	7411	7
7303	7412	7
7304	7413	7
7305	7414	7
7306	7415	7
7307	7416	7
7308	7417	7
7309	7418	7
7310	7419	7
7311	7420	7
7312	7421	8
7313	7422	8
7314	7423	8
7315	7424	8
7316	7425	8
7317	7426	9
7318	7427	9
7319	7428	9
7320	7429	9
7321	7430	10
7322	7431	10

```
!
! SIGNAL(DBG$NOTASTRUCT, 1, LAST_OPERAND[TOKEN$B_LENGTH]);
! Append the last operand to
! the current Pathname Descriptor and convert that to a
! Primary Descriptor.
APPEND TO PATHNAME(PATHDESC, LAST_OPERAND, NOT REC_COMP);
PRIMPTR = PATHNAME_TO_PRIMARY(PATHDESC, SUBSCR_DESC,
                              .PLIPTR, SAVED_PATHDESC);
END; ! TEMP_BLOCK

GET_BLISS_SUBSCRIPTS(.PRIMPTR, LAST_OPERAND[TOKEN$B_LENGTH]);
END;

! Handler the PASCAL dereference operator (^). Append the
! last identifier onto the pathname. Then build the pathname
! into a Primary Descriptor. Then call the routine that
! handles dereferencing --it just lights the EVAL bit on the
! sub-node and then allocates a new subnode for the object
! being pointed to.
[PRIMARY$K_ACT_START_DEREF]:
BEGIN
  LABEL TEMP_BLOCK;

  ! If the last operand was an identifier, we append it to
  ! the current Pathname Descriptor and convert that to a
  ! Primary Descriptor.
  TEMP_BLOCK: BEGIN
    IF .LAST_OPERAND[TOKEN$W_CODE] EQL TOKEN$K_IDENTIFIER
    THEN
      BEGIN

        ! First check for DEFINED symbols.
        ! Check that no invocation number is present.
        IF .PATHDESC[PTH$B_LOCINVOC] EQL 0
        THEN

          ! Look up the symbol in the DEFINE symbol table.
          BEGIN
            IF DBG$DEF_SYM_FIND (LAST_OPERAND [TOKEN$B_LENGTH],
                                KIND, PRIMPTR,
                                DUMMY, DUMMY)
            THEN
              BEGIN
                IF .KIND EQL DEFINE_ADDRESS
                OR .KIND EQL DEFINE_VALUE
                THEN
                  BEGIN
```

```

7323 7432 10
7324 7433 10
7325 7434 10
7326 7435 10
7327 7436 10
7328 7437 10
7329 7438 10
7330 7439 10
7331 7440 9
7332 7441 8
7333 7442 7
7334 7443 6
7335 7444 6
7336 7445 6
7337 7446 6
7338 7447 5
7339 7448 5
7340 7449 5
7341 7450 4
7342 7451 4
7343 7452 4
7344 7453 4
7345 7454 4
7346 7455 4
7347 7456 4
7348 7457 4
7349 7458 4
7350 7459 4
7351 7460 4
7352 7461 5
7353 7462 5
7354 7463 5
7355 7464 5
7356 7465 5
7357 7466 5
7358 7467 5
7359 7468 5
7360 7469 6
7361 7470 5
7362 7471 6
7363 7472 6
7364 7473 6
7365 7474 6
7366 7475 6
7367 7476 5
7368 7477 5
7369 7478 5
7370 7479 5
7371 7480 5
7372 7481 5
7373 7482 5
7374 7483 5
7375 7484 5
7376 7485 4
7377 7486 4
7378 7487 4
7379 7488 4

```

```

! We have found a matching DEFINED symbol.
! Copy the descriptor into temporary memory.
! (Fourth parameter FALSE <-> copy into tempmem).
DBG$NCPY_DESC (.PRIMPTR, PRIMPTR,
               DUMMY, FALSE);
LEAVE TEMP_BLOCK;
END;
END;
END;
APPEND TO_PATHNAME(PATHDESC, .LAST_OPERAND, NOT REC_COMP);
PRIMPTR = PATHNAME_TO_PRIMARY(PATHDESC, SUBSCR_DESC,
                              .PLIPTR, SAVED_PATHDESC);
END; ! TEMP_BLOCK
GET_DEREFERENCE(.PRIMPTR);
END;

! Handle a PLI dereference as in A->B. Here we convert the
! left-hand-side to a Primary, and then save away the Primary.
! The parse is then "backed up" to the start state to pick
! up the right hand side of the "->". Later, in the
! PATHNAME_TO_PRIMARY routine, the two primaries are glued
! together.
[PRIMARY$K_ACT_START_DEREF_PLI]:
BEGIN
    ! Save away the Primary for the stuff to the left of "->".
    !
    APPEND TO_PATHNAME(PATHDESC, .LAST_OPERAND, NOT REC_COMP);
    PLIPTR = PATHNAME_TO_PRIMARY(PATHDESC, SUBSCR_DESC,
                                .PLIPTR, SAVED_PATHDESC);
    IF (.PLIPTR[DBG$B_DHDR_KIND] NEQ RST$K_DATA) OR
        (.PLIPTR[DBG$B_DHDR_FCODE] NEQ RST$K_TYPE_PTR)
    THEN
        BEGIN
            LOCAL
                NAME;
            DBG$NPATHDESC TO_CS(PATHDESC, NAME);
            SIGNAL(DBG$VALNOTADDR, 1, .NAME);
            END;

        ! Re-initialize.
        !
        PRIMPTR = 0;
        NUMERIC_PATHNAME = FALSE;
        CH$FILL TO_DBG$K_PATHNAME_SIZE * %UPVAL, PATHDESC);
        PATHVECTOR = PATHDESC[PTH$A_PATHVECTOR];
        CH$FILL(0, SUBSCR_DESC_SIZE, SUBSCR_DESC);
        END;

! Handle the built-in function operator that follows the

```

```
7380 7489 4
7381 7490 4
7382 7491 4
7383 7492 4
7384 7493 5
7385 7494 5
7386 7495 5
7387 7496 5
7388 7497 5
7389 7498 5
7390 7499 5
7391 7500 5
7392 7501 5
7393 7502 5
7394 7503 5
7395 7504 5
7396 7505 5
7397 7506 5
7398 7507 5
7399 7508 5
7400 7509 5
7401 7510 5
7402 7511 5
7403 7512 5
7404 7513 5
7405 7514 5
7406 7515 5
7407 7516 5
7408 7517 5
7409 7518 6
7410 7519 6
7411 7520 6
7412 7521 6
7413 7522 6
7414 7523 6
7415 7524 6
7416 7525 6
7417 7526 6
7418 7527 6
7419 7528 6
7420 7529 6
7421 7530 6
7422 7531 6
7423 7532 6
7424 7533 6
7425 7534 6
7426 7535 6
7427 7536 6
7428 7537 6
7429 7538 6
7430 7539 6
7431 7540 6
7432 7541 6
7433 7542 6
7434 7543 6
7435 7544 6
7436 7545 5
```

```
built-in function name. This will return a primary that
represents the value of the built-in function.
[PRIMARY$K_ACT_START_BIF_CALL]:
BEGIN

    ! If the last operand was not a built-in function name,
    ! signal the error.
    IF .LAST_OPERAND[TOKEN$B_CODE] NEQ TOKEN$K_BUILTIN_FUNCTION
    THEN
        SIGNAL(DBG$MISOPEMIS, 1, TOKEN[TOKEN$B_OPLEN]);

        ! Call a routine that returns a counted longword vector of
        ! pointers that point to primary tokens of the arguments
        ! of the specified built-in function call. The number of
        ! arguments expected in the B_FLAGS field of a built-in
        ! function operand.
        ARG_LIST = DBG$GET_BIF_ARGUMENTS(.LAST_OPERAND[TOKEN$B_FLAGS],
                                          LAST_OPERAND[TOKEN$B_LENGTH]);

        ! For now, we will not accept any built-in function that
        ! contains more than 2 arguments. If more than 2 are
        ! present, an invalid argument list message is signaled.
        IF .ARG_LIST[0] LEQ 2
        THEN
            BEGIN
                ! Create a new operator token to represent the specific
                ! built-in function call and then pass this token to
                ! DBG$EVAL_LANG_OPERATOR to process the built-in
                ! function and return its value as a primary.
                IF .ARG_LIST[0] EQL 1
                THEN
                    TOKEN = CREATE_OPERATOR_TOKEN(.LAST_OPERAND[TOKEN$B_BIF],
                                                    LAST_OPERAND[TOKEN$B_LENGTH],
                                                    .TOKEN[TOKEN$B_KIND])
                ELSE
                    TOKEN = CREATE_OPERATOR_TOKEN(.LAST_OPERAND[TOKEN$B_BIF],
                                                    LAST_OPERAND[TOKEN$B_LENGTH],
                                                    TOKEN$K_INFIX_OP);

                PRIMPTR = DBG$EVAL_LANG_OPERATOR(.TOKEN, .ARG_LIST[1], .ARG_LIST[2]);

                ! Return the fact that we have built the result by now
                ! and that it is a value descriptor. Because of the
                ! way the BIF tokens come through the above initialization
                ! code, this value was never set. And It Better Be!
                RET_OPERAND_FLAG[0] = TRUE;
            END
        ELSE
```



```
7437 7546 5
7438 7547 5
7439 7548 4
7440 7549 4
7441 7550 4
7442 7551 4
7443 7552 4
7444 7553 4
7445 7554 4
7446 7555 4
7447 7556 5
7448 7557 5
7449 7558 5
7450 7559 5
7451 7560 5
7452 7561 5
7453 7562 5
7454 7563 5
7455 7564 5
7456 7565 6
7457 7566 6
7458 7567 6
7459 7568 6
7460 7569 6
7461 7570 6
7462 7571 6
7463 7572 6
7464 7573 6
7465 7574 6
7466 7575 6
7467 7576 6
7468 7577 7
7469 7578 7
7470 7579 7
7471 7580 7
7472 7581 7
7473 7582 8
7474 7583 8
7475 7584 8
7476 7585 8
7477 7586 9
7478 7587 9
7479 7588 9
7480 7589 9
7481 7590 9
7482 7591 9
7483 7592 9
7484 7593 9
7485 7594 9
7486 7595 9
7487 7596 8
7488 7597 8
7489 7598 7
7490 7599 7
7491 7600 6
7492 7601 6
7493 7602 6
```

```
SIGNAL(DBG$_INVARGLIS, 1, LAST_OPERAND[TOKEN$B_LENGTH]);
EXITLOOP;
END;

! Handle the Ada tick operator immediately after the start of
! the symbol. The symbol thus consists of just a single type
! name.
[PRIMARY$K_ACT_START_TICK]:
BEGIN

! If the last operand was an identifier, we append it to
! the current Pathname Descriptor and convert that to a
! Primary Descriptor.
IF .LAST_OPERAND[TOKEN$W_CODE] EQL TOKEN$K_IDENTIFIER
THEN
BEGIN

! First check for DEFINED symbols.
! Check that no invocation number is present.
IF .PATHDESC[PTH$B_LOCINVO] EQL 0
THEN

! Look up the symbol in the DEFINE symbol table.
BEGIN
IF DBG$DEF_SYM_FIND (LAST_OPERAND [TOKEN$B_LENGTH],
KIND, PRIMPTR,
DUMMY, DUMMY)
THEN
BEGIN
IF .KIND EQL DEFINE_ADDRESS
OR .KIND EQL DEFINE_VALUE
THEN
BEGIN

! We have found a matching DEFINED symbol.
! Copy the descriptor into temporary memory.
! (Fourth parameter FALSE <-> copy into tempmem).
DBG$NCOPY_DESC (.PRIMPTR, PRIMPTR,
DUMMY, FALSE);
EXITLOOP;
END;

END;

END;

APPEND_TO_PATHNAME(PATHDESC, .LAST_OPERAND, NOT_REC_COMP);
```

```

7494
7495
7496
7497
7498
7499
7500
7501
7502
7503
7504
7505
7506
7507
7508
7509
7510
7511
7512
7513
7514
7515
7516
7517
7518
7519
7520
7521
7522
7523
7524
7525
7526
7527
7528
7529
7530
7531
7532
7533
7534
7535
7536
7537
7538
7539
7540
7541
7542
7543
7544
7545
7546
7547
7548
7549
7550

```

```

7603 6
7604 6
7605 6
7606 6
7607 6
7608 6
7609 6
7610 6
7611 6
7612 6
7613 6
7614 6
7615 7
7616 7
7617 7
7618 7
7619 7
7620 7
7621 7
7622 7
7623 7
7624 7
7625 7
7626 6
7627 6
7628 6
7629 6
7630 6
7631 6
7632 6
7633 6
7634 6
7635 6
7636 6
7637 6
7638 6
7639 5
7640 5
7641 5
7642 5
7643 4
7644 4
7645 4
7646 4
7647 4
7648 4
7649 4
7650 4
7651 5
7652 5
7653 5
7654 5
7655 5
7656 5
7657 5
7658 5
7659 5

```

```

: Call GETSYMBOL directly since all we need is the typeid.
: TRUE is passed in to tell GETSYMBOL that we do want it
: to look up symbol-types as well as the normal data-types.
DBGSTA_GETSYMBOL(PATHDESC, TYPEID, KIND, 0, 0, 0, TRUE);
: Check the output from getsymbol to see if a unique symbol
: was found. If not, signal the appropriate error.
IF .TYPEID EQL 0
THEN
  BEGIN
    DBG$NPATHDESC TO CS(PATHDESC, PATHSTRING);
    IF .KIND EQL RST$K_NOTUNIQUE
    THEN
      SIGNAL(DBG$_NOUNIQUE, 1, .PATHSTRING)
    ELSE
      IF .KIND EQL RST$K_OVERLOAD
      THEN
        SIGNAL(DBG$_NOTUNQOVR, 1, .PATHSTRING)
      ELSE
        SIGNAL(DBG$_NOSYMBOL, 1, .PATHSTRING);
    END;
    PRIMPTR = DBG$EVAL_ADA_TICK(.TYPEID, .TOKEN);
    : Return the fact that we have built the result by now
    : and that it is a value descriptor. Because of the
    : way the Ada tick tokens come through the above
    : initialization code, this value was never set.
    : And It Better Be!
    RET_OPERAND_FLAG[0] = TRUE;
  END
ELSE
  SIGNAL(DBG$_SYNERREXPR, 1, .LAST_OPERAND[TOKEN$B_LENGTH]);
EXITLOOP;
END;
: Handle the terminator operator immediately after the start of
: the symbol. The symbol thus consists of just a single name
: or a constant.
[PRIMARY$K_ACT_START_TERM]:
  BEGIN
    : If the last operand was an identifier, we append it to
    : the current Pathname Descriptor and convert that to a
    : Primary Descriptor.
    IF .LAST_OPERAND[TOKEN$B_CODE] EQL TOKEN$K_IDENTIFIER
    THEN

```

```

7551 7660 6
7552 7661 6
7553 7662 6
7554 7663 6
7555 7664 6
7556 7665 6
7557 7666 6
7558 7667 6
7559 7668 6
7560 7669 6
7561 7670 6
7562 7671 6
7563 7672 7
7564 7673 7
7565 7674 7
7566 7675 7
7567 7676 7
7568 7677 8
7569 7678 8
7570 7679 8
7571 7680 8
7572 7681 9
7573 7682 9
7574 7683 9
7575 7684 9
7576 7685 9
7577 7686 9
7578 7687 9
7579 7688 9
7580 7689 9
7581 7690 9
7582 7691 8
7583 7692 8
7584 7693 7
7585 7694 7
7586 7695 6
7587 7696 6
7588 7697 6
7589 7698 6
7590 7699 6
7591 7700 6
7592 7701 6
7593 7702 6
7594 7703 6
7595 7704 6
7596 7705 6
7597 7706 5
7598 7707 5
7599 7708 5
7600 7709 5
7601 7710 5
7602 7711 5
7603 7712 5
7604 7713 4
7605 7714 4
7606 7715 4
7607 7716 4

```

```

BEGIN

! First check for DEFINED symbols.
! Check that no invocation number is present.
IF .PATHDESC[PTHSB_LOCINVO] EQL 0
THEN

! Look up the symbol in the DEFINE symbol table.
BEGIN
IF DBGSDEF_SYM_FIND (LAST_OPERAND [TOKENSB_LENGTH],
                     KIND, PRIMPTR,
                     DUMMY, DUMMY)
THEN
BEGIN
IF .KIND EQL DEFINE_ADDRESS
OR .KIND EQL DEFINE_VALUE
THEN
BEGIN

! We have found a matching DEFINED symbol.
! Copy the descriptor into temporary memory.
! (fourth parameter FALSE <-> copy into tempmem).
DBGSNCOPY_DESC (.PRIMPTR, PRIMPTR,
                DUMMY, FALSE);
EXITLOOP;
END;

END;

END;

APPEND TO PATHNAME(PATHDESC, .LAST_OPERAND, NOT REC_COMP);
PRIMPTR = PATHNAME_TO_PRIMARY(PATHDESC, SUBSCR_DESC,
                              .PLIPTR, SAVED_PATHDESC);
END

! Otherwise, the last operand was a constant of some
! sort so we create a Value Descriptor for it.
ELSE
PRIMPTR = CONSTANT_TO_VALDESCR(.LAST_OPERAND);

! Exit from the parse loop (the get-token loop).
EXITLOOP;
END;

! Handle the backslash operator after a previous backslash ope-

```

```
7608 7717 4
7609 7718 4
7610 7719 4
7611 7720 4
7612 7721 4
7613 7722 4
7614 7723 4
7615 7724 4
7616 7725 4
7617 7726 4
7618 7727 4
7619 7728 4
7620 7729 4
7621 7730 4
7622 7731 4
7623 7732 4
7624 7733 4
7625 7734 4
7626 7735 4
7627 7736 4
7628 7737 5
7629 7738 5
7630 7739 5
7631 7740 5
7632 7741 4
7633 7742 4
7634 7743 4
7635 7744 4
7636 7745 4
7637 7746 4
7638 7747 4
7639 7748 4
7640 7749 5
7641 7750 5
7642 7751 4
7643 7752 4
7644 7753 4
7645 7754 4
7646 7755 4
7647 7756 4
7648 7757 4
7649 7758 4
7650 7759 4
7651 7760 5
7652 7761 5
7653 7762 5
7654 7763 5
7655 7764 5
7656 7765 4
7657 7766 4
7658 7767 4
7659 7768 4
7660 7769 4
7661 7770 4
7662 7771 4
7663 7772 4
7664 7773 5
```

```
! rator (e.g., A\B\C). Note that this is not allowed after a
! numeric pathname. Also handle a backslash operator after an
! invocation number. Append the last operand (which must be an
! identifier) to the current Pathname Descriptor.
```

```
[PRIMARYSK_ACT_SLASH_SLASH]:
BEGIN
  IF .NUMERIC_PATHNAME
  THEN
    SIGNAL(DBG$_ILLPATHELEM, 1, LAST_OPERAND[TOKEN$B_LENGTH]);

    APPEND_TO_PATHNAME(PATHDESC, .LAST_OPERAND, NOT_REC_COMP);
  END;
```

```
! Handle a dot (data qualification) after a backslash. Here we
! complete the current Pathname Descriptor and convert it to a
! Primary Descriptor.
```

```
[PRIMARYSK_ACT_SLASH_DOT]:
BEGIN
  APPEND_TO_PATHNAME(PATHDESC, .LAST_OPERAND, NOT_REC_COMP);
  PRIMPTR = PATHNAME_TO_PRIMARY(PATHDESC, SUBSCR_DESC,
    .PLIPTR, SAVED_PATHDESC);
END;
```

```
! Handle a dot after a backslash. In PLI we do not call
! PATHNAME_TO_PRIMARY until after collecting all the record
! components.
```

```
[PRIMARYSK_ACT_SLASH_DOT_PLI]:
BEGIN
  APPEND_TO_PATHNAME(PATHDESC, .LAST_OPERAND, NOT_REC_COMP);
END;
```

```
! Handle a subscript parenthesis after a backslash. Here we
! complete the current Pathname Descriptor and convert it to a
! Primary Descriptor. We then pick up all the subscript expres-
! sions.
```

```
[PRIMARYSK_ACT_SLASH_SUBSCR]:
BEGIN
  APPEND_TO_PATHNAME(PATHDESC, .LAST_OPERAND, NOT_REC_COMP);
  PRIMPTR = PATHNAME_TO_PRIMARY(PATHDESC, SUBSCR_DESC,
    .PLIPTR, SAVED_PATHDESC, TRUE);
  GET_SUBSCRIPTS(.PRIMPTR);
END;
```

```
! Handle a subscript after a backslash in language PLI. In PLI
! we do not incorporate the subscripts into the Primary
! Descriptor until later.
```

```
[PRIMARYSK_ACT_SLASH_SUBSCR_PLI]:
BEGIN
```


7665	7774	5
7666	7775	5
7667	7776	4
7668	7777	4
7669	7778	4
7670	7779	4
7671	7780	4
7672	7781	4
7673	7782	4
7674	7783	4
7675	7784	4
7676	7785	4
7677	7786	5
7678	7787	5
7679	7788	5
7680	7789	5
7681	7790	5
7682	7791	4
7683	7792	4
7684	7793	4
7685	7794	4
7686	7795	4
7687	7796	4
7688	7797	4
7689	7798	4
7690	7799	4
7691	7800	4
7692	7801	4
7693	7802	5
7694	7803	5
7695	7804	5
7696	7805	5
7697	7806	5
7698	7807	4
7699	7808	4
7700	7809	4
7701	7810	4
7702	7811	4
7703	7812	4
7704	7813	4
7705	7814	4
7706	7815	4
7707	7816	4
7708	7817	4
7709	7818	5
7710	7819	5
7711	7820	5
7712	7821	5
7713	7822	5
7714	7823	5
7715	7824	5
7716	7825	5
7717	7826	6
7718	7827	5
7719	7828	6
7720	7829	6
7721	7830	6

```
APPEND_TO_PATHNAME(PATHDESC, .LAST_OPERAND, NOT_REC_COMP);  
SAVE_SUBSCRIPTS(PATHDESC, SUBSCR_DESC);  
END;
```

```
! Handle a subscript left bracket after a backslash for language  
! BLISS. Here we complete the current Pathname Descriptor and  
! convert it to a Primary Descriptor. We then pick up all the  
! subscript expressions. BLISS has to be treated separately  
! because it represents data differently in the RST and DST.
```

```
[PRIMARY$K_ACT_SLASH_SUBSCR_BLI]:  
BEGIN  
APPEND_TO_PATHNAME(PATHDESC, .LAST_OPERAND, NOT_REC_COMP);  
PRIMPTR = PATHNAME_TO_PRIMARY(PATHDESC, SUBSCR_DESC,  
                                .PLIPTR, SAVED_PATHDESC);  
GET_BLISS_SUBSCRIPTS(.PRIMPTR, LAST_OPERAND[TOKEN$B_LENGTH]);  
END;
```

```
! Handle the PASCAL dereference operator (^) occurring after  
! a pathname qualified name. Append the last identifier onto  
! the pathname. Then build the pathname into a Primary  
! Descriptor. Then call the routine that handles dereferencing  
! --it just lights the EVAL bit on the sub-node and then  
! allocates a new subnode for the object being pointed to.
```

```
[PRIMARY$K_ACT_SLASH_DEREF]:  
BEGIN  
APPEND_TO_PATHNAME(PATHDESC, .LAST_OPERAND, NOT_REC_COMP);  
PRIMPTR = PATHNAME_TO_PRIMARY(PATHDESC, SUBSCR_DESC,  
                                .PLIPTR, SAVED_PATHDESC);  
GET_DEREFERENCE(.PRIMPTR);  
END;
```

```
! Handle a PLI dereference as in A\B->C. Here we convert the  
! left-hand-side to a Primary, and then save away the Primary.  
! The parse is then "backed up" to the start state to pick  
! up the right hand side of the "->". Later, in the  
! PATHNAME_TO_PRIMARY routine, the two primaries are glued  
! together.
```

```
[PRIMARY$K_ACT_SLASH_DEREF_PLI]:  
BEGIN  
! Obtain value of expression to left of "->".  
APPEND_TO_PATHNAME(PATHDESC, .LAST_OPERAND, NOT_REC_COMP);  
PLIPTR = PATHNAME_TO_PRIMARY(PATHDESC, SUBSCR_DESC,  
                              .PLIPTR, SAVED_PATHDESC);  
IF (.PLIPTR[DBG$B_DHDR_KIND] NEQ RST$K_DATA) OR  
   (.PLIPTR[DBG$B_DHDR_FCODE] NEQ RST$K_TYPE_PTR)  
THEN  
BEGIN  
LOCAL  
NAME;
```

```
7722 7831 6
7723 7832 6
7724 7833 6
7725 7834 6
7726 7835 6
7727 7836 6
7728 7837 6
7729 7838 6
7730 7839 6
7731 7840 6
7732 7841 6
7733 7842 6
7734 7843 6
7735 7844 6
7736 7845 6
7737 7846 6
7738 7847 6
7739 7848 6
7740 7849 6
7741 7850 6
7742 7851 6
7743 7852 6
7744 7853 6
7745 7854 6
7746 7855 6
7747 7856 6
7748 7857 6
7749 7858 6
7750 7859 6
7751 7860 6
7752 7861 6
7753 7862 6
7754 7863 6
7755 7864 6
7756 7865 6
7757 7866 6
7758 7867 6
7759 7868 6
7760 7869 6
7761 7870 6
7762 7871 6
7763 7872 6
7764 7873 6
7765 7874 6
7766 7875 6
7767 7876 6
7768 7877 6
7769 7878 6
7770 7879 6
7771 7880 6
7772 7881 6
7773 7882 6
7774 7883 6
7775 7884 6
7776 7885 6
7777 7886 6
7778 7887 6
```

```
DBG$NPATHDESC TO CS(PATHDESC, NAME);
SIGNAL(DBG$_VALNOTADDR, 1, .NAME);
END;
```

```
! Re-initialize.
```

```
PRIMPTR = 0;
NUMERIC_PATHNAME = FALSE;
CH$FILLTO, DBG$K_PATHNAME SIZE*ZUPVAL, PATHDESC);
PATHVECTOR = PATHDESC[PTH$A_PATHVECTOR];
CH$FILL(0, SUBSCR_DESC_SIZE, SUBSCR_DESC);
END;
```

```
! Handle the Ada tick operator after a backslash. Here we
! just complete the Pathname Descriptor and convert it to a
! Primary Descriptor. We are then done with the symbol.
```

```
[PRIMARY$K_ACT_SLASH_TICK]:
```

```
BEGIN
APPEND_TO_PATHNAME(PATHDESC, .LAST_OPERAND, NOT_REC_COMP);

! Call GETSYMBOL directly since all we need is the typeid.
! TRUE is passed in to tell GETSYMBOL that we do want it
! to look up symbol-types as well as the normal data-types.
```

```
DBG$STA_GETSYMBOL(PATHDESC, TYPEID, KIND, 0, 0, 0, TRUE);
```

```
! Check the output from getsymbol to see if a unique symbol
! was found. If not, signal the appropriate error.
```

```
IF .TYPEID EQL 0
THEN
```

```
BEGIN
DBG$NPATHDESC TO CS(PATHDESC, PATHSTRING);
IF .KIND EQL RST$K_NOTUNIQUE
THEN
SIGNAL(DBG$_NOUNIQUE, 1, .PATHSTRING)
ELSE
IF .KIND EQL RST$K_OVERLOAD
THEN
SIGNAL(DBG$_NOTUNQOVR, 1, .PATHSTRING)
ELSE
SIGNAL(DBG$_NOSYMBOL, 1, .PATHSTRING);
END;
```

```
PRIMPTR = DBG$EVAL_ADA_TICK(.TYPEID, .TOKEN);
```

```
! Return the fact that we have built the result by now
! and that it is a value descriptor. Because of the
! way the Ada tick tokens come through the above
! initialization code, this value was never set.
! And It Better Be!
```

```
RET_OPERAND_FLAG[0] = TRUE;
```

```
EXITLOOP;
```

7779	7888	4	END;
7780	7889	4	
7781	7890	4	
7782	7891	4	! Handle the terminator operator after a backslash. Here we
7783	7892	4	just complete the Pathname Descriptor and convert it to a
7784	7893	4	Primary Descriptor. We are then done with the symbol.
7785	7894	4	
7786	7895	4	[PRIMARYSK_ACT_SLASH_TERM]:
7787	7896	5	BEGIN
7788	7897	5	APPEND_TO_PATHNAME(PATHDESC, .LAST_OPERAND, NOT_REC_COMP);
7789	7898	5	PRIMPTR = PATHNAME_TO_PRIMARY(PATHDESC, SUBSCR_DESC,
7790	7899	5	.PLI_PTR, SAVED_PATHDESC);
7791	7900	5	EXITLOOP;
7792	7901	4	END;
7793	7902	4	
7794	7903	4	
7795	7904	4	! Handle a slash after an 'OF' in COBOL.
7796	7905	4	
7797	7906	4	[PRIMARYSK_ACT_DOT_SLASH_COB]:
7798	7907	5	BEGIN
7799	7908	5	APPEND_TO_PATHNAME(PATHDESC, .LAST_OPERAND, NOT_REC_COMP);
7800	7909	4	END;
7801	7910	4	
7802	7911	4	! Handle a dot (data qualification) after another dot. Check
7803	7912	4	that the last operand is an identifier and that it is a valid
7804	7913	4	component name of the current record type. Get its SYMID,
7805	7914	4	etc., and add it to the Primary Descriptor being built.
7806	7915	4	
7807	7916	4	[PRIMARYSK_ACT_DOT_DOT]:
7808	7917	4	GET_RECORD_COMPONENT(.PRIMPTR, LAST_OPERAND[TOKEN\$B_LENGTH]);
7809	7918	4	
7810	7919	4	
7811	7920	4	! Handle a dot after another dot in PLI. In PLI we do not
7812	7921	4	call PATHNAME_TO_PRIMARY until after collecting all
7813	7922	4	the record components.
7814	7923	4	
7815	7924	4	[PRIMARYSK_ACT_DOT_DOT_PLI]:
7816	7925	5	BEGIN
7817	7926	5	APPEND_TO_PATHNAME(PATHDESC, .LAST_OPERAND, REC_COMP);
7818	7927	4	END;
7819	7928	4	
7820	7929	4	
7821	7930	4	! Handle an 'OF' operator after another 'OF' operator :
7822	7931	4	COBOL.
7823	7932	4	
7824	7933	4	[PRIMARYSK_ACT_DOT_DOT_COB]:
7825	7934	5	BEGIN
7826	7935	5	APPEND_TO_PATHNAME(PATHDESC, .LAST_OPERAND, COB_REC_COMP);
7827	7936	4	END;
7828	7937	4	
7829	7938	4	! Handle subscripting after a dot (data qualification). Check
7830	7939	4	that the last operand is an identifier and that it is a valid
7831	7940	4	component name of the current record type. Get its SYMID,
7832	7941	4	etc., and add it to the Primary Descriptor being built. Then
7833	7942	4	pick up all the subscript expressions and add their values to
7834	7943	4	the Primary Descriptor.
7835	7944	4	

```
7836 7945 4 [PRIMARYSK_ACT_DOT_SUBSCR]:
7837 7946 5 BEGIN
7838 7947 5 GET_RECORD_COMPONENT(.PRIMPTR, LAST_OPERAND[TOKEN$B_LENGTH]);
7839 7948 5 GET_SUBSCRIPTS(.PRIMPTR);
7840 7949 4 END;
7841 7950 4
7842 7951 4
7843 7952 4 ! Handle a subscript after a dot in PLI. In PLI we do not
7844 7953 4 call PATHNAME_TO_PRIMARY until after picking up all
7845 7954 4 the record components.
7846 7955 4
7847 7956 4 [PRIMARYSK_ACT_DOT_SUBSCR_PLI]:
7848 7957 5 BEGIN
7849 7958 5 APPEND_TO_PATHNAME(PATHDESC, .LAST_OPERAND, REC_COMP);
7850 7959 5 SAVE_SUBSCRIPTS(PATHDESC, SUBSCR_DESC);
7851 7960 4 END;
7852 7961 4
7853 7962 4
7854 7963 4 ! Handle the subscript operator after the 'OF' operator
7855 7964 4 in COBOL.
7856 7965 4
7857 7966 4 [PRIMARYSK_ACT_DOT_SUBSCR_COB]:
7858 7967 5 BEGIN
7859 7968 5 APPEND_TO_PATHNAME(PATHDESC, .LAST_OPERAND, NOT_REC_COMP);
7860 7969 5 SAVE_SUBSCRIPTS(PATHDESC, SUBSCR_DESC);
7861 7970 4 END;
7862 7971 4
7863 7972 4
7864 7973 4 ! Handle the PASCAL dereference operator (^) occurring after
7865 7974 4 a dot (data qualification). Here we call a routine which
7866 7975 4 appends the last operand (representing a record component)
7867 7976 4 onto the Primary Descriptor being built. We then call the
7868 7977 4 routine that handles dereferencing--it just lights the EVAL
7869 7978 4 bit on the sub-node and then allocates a new subnode for
7870 7979 4 the object being pointed to.
7871 7980 4
7872 7981 5 [PRIMARYSK_ACT_DOT_DEREF]:
7873 7982 5 BEGIN
7874 7983 5 GET_RECORD_COMPONENT(.PRIMPTR, LAST_OPERAND[TOKEN$B_LENGTH]);
7875 7984 5 GET_DEREFERENCE(.PRIMPTR);
7876 7985 4 END;
7877 7986 4
7878 7987 4
7879 7988 4 ! Handle a PLI dereference as in A.B->C. Here we convert the
7880 7989 4 left-hand-side to a Primary, and then save away the Primary.
7881 7990 4 We loop back to the start state to pick up the rest
7882 7991 4 of the expression.
7883 7992 4
7884 7993 5 [PRIMARYSK_ACT_DOT_DEREF_PLI]:
7885 7994 5 BEGIN
7886 7995 5 ! Save away the Primary constructed so far.
7887 7996 5
7888 7997 5 APPEND_TO_PATHNAME(PATHDESC, .LAST_OPERAND, REC_COMP);
7889 7998 5 PLIPTR = PATHNAME_TO_PRIMARY(PATHDESC, SUBSCR_DESC,
7890 7999 5 .PLIPTR, SAVED_PATHDESC);
7891 8000 5 IF (.PLIPTR[DBG$B_DHDR_KIND] NEQ RST$K_DATA) OR
7892 8001 6 (.PLIPTR[DBG$B_DHDR_FCODE] NEQ RST$K_TYPE_PTR)
```



```

7893 8002 5
7894 8003 6
7895 8004 6
7896 8005 6
7897 8006 6
7898 8007 6
7899 8008 6
7900 8009 6
7901 8010 6
7902 8011 6
7903 8012 6
7904 8013 6
7905 8014 6
7906 8015 6
7907 8016 6
7908 8017 6
7909 8018 6
7910 8019 6
7911 8020 6
7912 8021 6
7913 8022 6
7914 8023 6
7915 8024 6
7916 8025 6
7917 8026 6
7918 8027 6
7919 8028 6
7920 8029 6
7921 8030 6
7922 8031 6
7923 8032 6
7924 8033 6
7925 8034 6
7926 8035 6
7927 8036 6
7928 8037 6
7929 8038 6
7930 8039 6
7931 8040 6
7932 8041 6
7933 8042 6
7934 8043 6
7935 8044 6
7936 8045 6
7937 8046 6
7938 8047 6
7939 8048 6
7940 8049 6
7941 8050 6
7942 8051 6
7943 8052 6
7944 8053 6
7945 8054 6
7946 8055 6
7947 8056 6
7948 8057 6
7949 8058 6

```

```

THEN
  BEGIN
    LOCAL
      NAME;
    DBG$NPATHDESC TO CS(PATHDESC, NAME);
    SIGNAL(DBG$_VALNOTADDR, 1, .NAME);
  END;

  ! Re-initialize.
  PRIMPTR = 0;
  NUMERIC PATHNAME = FALSE;
  CH$FILLTO, DBG$K_PATHNAME, SIZE*ZUPVAL, PATHDESC);
  PATHVECTOR = PATHDESC[PTH$A_PATHVECTOR];
  CH$FILL(0, SUBSCR_DESC_SIZE, SUBSCR_DESC);
END;

```

```

! Handle the Ada tick operator after a dot (data qualifica-
! tion). Call the routine that looks up the SYMID for the
! record component and adds that to the Primary Descriptor.
! Then exit the parse loop.
[PRIMARY$K_ACT_DOT_TICK]:
  BEGIN
    APPEND_TO_PATHNAME(PATHDESC, .LAST_OPERAND, REC_COMP);

    ! Call GETSYMBOL directly since all we need is the typeid.
    ! TRUE is passed in to tell GETSYMBOL that we do want it
    ! to look up symbol-types as well as the normal data-types.
    DBG$STA_GETSYMBOL(PATHDESC, TYPEID, KIND, 0, 0, 0, TRUE);

    ! Check the output from getsymbol to see if a unique symbol
    ! was found. If not, signal the appropriate error.
    IF .TYPEID EQL 0
    THEN
      BEGIN
        DBG$NPATHDESC TO CS(PATHDESC, PATHSTRING);
        IF .KIND EQL RST$K_NOTUNIQUE
        THEN
          SIGNAL(DBG$_NOUNIQUE, 1, .PATHSTRING)
        ELSE
          IF .KIND EQL RST$K_OVERLOAD
          THEN
            SIGNAL(DBG$_NOTUNQOVR, 1, .PATHSTRING)
          ELSE
            SIGNAL(DBG$_NOSYMBOL, 1, .PATHSTRING);
          END;
        PRIMPTR = DBG$EVAL_ADA_TICK(.TYPEID, .TOKEN);

        ! Return the fact that we have built the result by now
        ! and that it is a value descriptor. Because of the
        ! way the Ada tick tokens come through the above
        ! initialization code, this value was never set.

```

```

7950      8059 5      ! And It Better Be!
7951      8060 5      !
7952      8061 5      RET_OPERAND_FLAG[0] = TRUE;
7953      8062 5
7954      8063 5
7955      8064 4      EXITLOOP;
7956      8065 4      END;
7957      8066 4
7958      8067 4
7959      8068 4
7960      8069 4
7961      8070 4
7962      8071 4
7963      8072 4
7964      8073 5      [PRIMARY$K_ACT_DOT_TERM]:
7965      8074 5      BEGIN
7966      8075 5      GET_RECORD_COMPONENT(.PRIMPTR, LAST_OPERAND[TOKEN$B_LENGTH]);
7967      8076 4      EXITLOOP;
7968      8077 4      END;
7969      8078 4
7970      8079 4
7971      8080 4
7972      8081 4
7973      8082 4
7974      8083 5      [PRIMARY$K_ACT_DOT_TERM_PLI]:
7975      8084 5      BEGIN
7976      8085 5      APPEND_TO_PATHNAME(PATHDESC, .LAST_OPERAND, REC_COMP);
7977      8086 5      PRIMPTR = PATHNAME_TO_PRIMARY(PATHDESC, SUBSCR_DESC,
7978      8087 5      .PLIPTR, SAVED_PATHDESC);
7979      8088 4      EXITLOOP;
7980      8089 4      END;
7981      8090 4
7982      8091 4
7983      8092 4
7984      8093 4
7985      8094 4
7986      8095 5      [PRIMARY$K_ACT_DOT_TERM_COB]:
7987      8096 5      BEGIN
7988      8097 5      APPEND_TO_PATHNAME(PATHDESC, .LAST_OPERAND, NOT_REC_COMP);
7989      8098 5      PRIMPTR = PATHNAME_TO_PRIMARY(PATHDESC, SUBSCR_DESC,
7990      8099 5      .PLIPTR, SAVED_PATHDESC);
7991      8100 4      EXITLOOP;
7992      8101 4      END;
7993      8102 4
7994      8103 4
7995      8104 4
7996      8105 4
7997      8106 4
7998      8107 4
7999      8108 4
8000      8109 4
8001      8110 4
8002      8111 4
8003      8112 4
8004      8113 4
8005      8114 4
8006      8115 4

      ! Handle the terminator operator after a dot (data qualifica-
      ! tion). Call the routine that looks up the SYMID for the
      ! record component and adds that to the Primary Descriptor.
      ! Then exit the parse loop.

      [PRIMARY$K_ACT_DOT_TERM]:
      BEGIN
      GET_RECORD_COMPONENT(.PRIMPTR, LAST_OPERAND[TOKEN$B_LENGTH]);
      EXITLOOP;
      END;

      ! Handle the end of the Primary after a dot in PLI. At this
      ! point we call PATHNAME_TO_PRIMARY, and then exit the loop.

      [PRIMARY$K_ACT_DOT_TERM_PLI]:
      BEGIN
      APPEND_TO_PATHNAME(PATHDESC, .LAST_OPERAND, REC_COMP);
      PRIMPTR = PATHNAME_TO_PRIMARY(PATHDESC, SUBSCR_DESC,
      .PLIPTR, SAVED_PATHDESC);
      EXITLOOP;
      END;

      ! Handle the end of the Primary after an 'OF' operator
      ! (record component selection) in COBOL.

      [PRIMARY$K_ACT_DOT_TERM_COB]:
      BEGIN
      APPEND_TO_PATHNAME(PATHDESC, .LAST_OPERAND, NOT_REC_COMP);
      PRIMPTR = PATHNAME_TO_PRIMARY(PATHDESC, SUBSCR_DESC,
      .PLIPTR, SAVED_PATHDESC);
      EXITLOOP;
      END;

      ! Handle a dot (data qualification) after subscripting. We do
      ! nothing with the dot until after we pick up the component
      ! name. However, we do call a routine that fixes up the
      ! bounds in the array subnode after subscripting has been
      ! completed.

      [PRIMARY$K_ACT_SUBSCR_DOT]:
      FIX_UP_PRIMARY(.PRIMPTR);

      ! Handle a dot after a subscript in PLI. There is actually
      ! nothing to do here.

      [PRIMARY$K_ACT_SUBSCR_DOT_PLI]:
```

8007	8116	5	BEGIN
8008	8117	5	0;
8009	8118	4	END;
8010	8119	4	
8011	8120	4	
8012	8121	4	! Handle subscripting parentheses immediately after a previous
8013	8122	4	instance of subscripting. Just do the subscripting normally.
8014	8123	4	Pick up the subscript expressions and add their values to the
8015	8124	4	Primary Descriptor being built.
8016	8125	4	
8017	8126	4	[PRIMARY\$K_ACT_SUBSCR_SUBSCR]:
8018	8127	4	GET_SUBSCRIPTS(.PRIMPTR);
8019	8128	4	
8020	8129	4	
8021	8130	4	! Handle a subscript immediately after another subscript
8022	8131	4	in PLI. In PLI we save up the subscripts until after we
8023	8132	4	collect a full pathname including record components.
8024	8133	4	
8025	8134	4	[PRIMARY\$K_ACT_SUBSCR_SUBSCR_PLI]:
8026	8135	5	BEGIN
8027	8136	5	SAVE_SUBSCRIPTS(PATHDESC, SUBSCR_DESC);
8028	8137	4	END;
8029	8138	4	
8030	8139	4	
8031	8140	4	! Handle a PASCAL dereference operator (^) occurring after
8032	8141	4	a subscripting operation. First call a routine that fixes
8033	8142	4	up the bounds in the array subnode after subscripting has
8034	8143	4	been completed. Then call the routine that handles
8035	8144	4	dereferencing--it just lights the EVAL bit on the sub-node and
8036	8145	4	then allocates a new subnode for the object being pointed to.
8037	8146	4	
8038	8147	4	[PRIMARY\$K_ACT_SUBSCR_DEREF]:
8039	8148	5	BEGIN
8040	8149	5	FIX_UP_PRIMARY(.PRIMPTR);
8041	8150	5	GET_DEREFERENCE(.PRIMPTR);
8042	8151	4	END;
8043	8152	4	
8044	8153	4	
8045	8154	4	! Handle a PLI dereference as in A(2)->B. Here we convert the
8046	8155	4	left-hand-side to a Primary, and save away that Primary.
8047	8156	4	We loop back to the start state to pick up the rest
8048	8157	4	of the expression.
8049	8158	4	
8050	8159	4	[PRIMARY\$K_ACT_SUBSCR_DEREF_PLI]:
8051	8160	5	BEGIN
8052	8161	5	
8053	8162	5	! Save away the Primary obtained so far.
8054	8163	5	!
8055	8164	5	PLIPTR = PATHNAME_TO_PRIMARY(PATHDESC, SUBSCR_DESC,
8056	8165	5	.PLIPTR, SAVED_PATHDESC);
8057	8166	5	IF (.PLIPTR[DBG\$B_DHDR_KIND] NEQ RST\$K_DATA) OR
8058	8167	6	(.PLIPTR[DBG\$B_DHDR_FCODE] NEQ RST\$K_TYPE_PTR)
8059	8168	5	THEN
8060	8169	6	BEGIN
8061	8170	6	LOCAL
8062	8171	6	NAME;
8063	8172	6	DBG\$NPATHDESC_TO_CS(PATHDESC, NAME);

```
8064 8173 6
8065 8174 3
8066 8175 3
8067 8176 3
8068 8177 3
8069 8178 3
8070 8179 3
8071 8180 3
8072 8181 3
8073 8182 3
8074 8183 4
8075 8184 4
8076 8185 4
8077 8186 4
8078 8187 4
8079 8188 4
8080 8189 4
8081 8190 4
8082 8191 4
8083 8192 4
8084 8193 3
8085 8194 3
8086 8195 3
8087 8196 4
8088 8197 4
8089 8198 4
8090 8199 4
8091 8200 4
8092 8201 4
8093 8202 4
8094 8203 3
8095 8204 3
8096 8205 3
8097 8206 3
8098 8207 4
8099 8208 4
8100 8209 4
8101 8210 4
8102 8211 4
8103 8212 4
8104 8213 4
8105 8214 4
8106 8215 4
8107 8216 4
8108 8217 4
8109 8218 4
8110 8219 4
8111 8220 4
8112 8221 4
8113 8222 4
8114 8223 4
8115 8224 4
8116 8225 4
8117 8226 4
8118 8227 4
8119 8228 4
8120 8229 4
```

```
SIGNAL(DBG$_VALNOTADDR, 1, .NAME);
END;
```

```
! Re-initialize.
```

```
PRIMPTR = 0;
NUMERIC_PATHNAME = FALSE;
CH$FILLTO, DBG$K_PATHNAME, SIZE*%UPVAL, PATHDESC);
PATHVECTOR = PATHDESC[PATH$A_PATHVECTOR];
CH$FILL(0, SUBSCR_DESC_SIZE, SUBSCR_DESC);
END;
```

```
! Handle subscripting followed by the terminator operator.
! First call a routine that fixes up the bounds in the array
! subnode after subscripting has been completed. The
! Primary Descriptor is now complete, so we exit the parse
! loop.
```

```
[PRIMARY$K_ACT_SUBSCR_TERM]:
BEGIN
FIX UP PRIMARY(.PRIMPTR);
EXITLOOP;
END;
```

```
! Handle a subscript followed by the end of the expression
! in PLI.
```

```
[PRIMARY$K_ACT_SUBSCR_TERM_PLI]:
BEGIN
PRIMPTR = PATHNAME_TO_PRIMARY(PATHDESC, SUBSCR_DESC,
.PLI$PTR, SAVED_PATHDESC);
EXITLOOP;
END;
```

```
! Handle a PASCAL dereference operator (^) followed by a dot.
! (E.g. "A^.B"). We actually do nothing here since the deref-
! erence operator was handled already, and the dot operator is
! not handled until after we pick up the component.
```

```
[PRIMARY$K_ACT_DEREF_DOT]:
0;
```

```
! Handle a PASCAL dereference operator (^) followed by a
! subscript. Here we just pick up the subscripts.
```

```
[PRIMARY$K_ACT_DEREF_SUBSCR]:
GET_SUBSCRIPTS(.PRIMPTR);
```

```
! Handle a PASCAL dereference operator (^) followed by another
! dereference. We call the routine that handles dereferencing--
! it just lights the EVAL bit on the sub-node and then allo-
```



```

: 8121      8230      4
: 8122      8231      4
: 8123      8232      4
: 8124      8233      4
: 8125      8234      4
: 8126      8235      4
: 8127      8236      4
: 8128      8237      4
: 8129      8238      4
: 8130      8239      4
: 8131      8240      4
: 8132      8241      4
: 8133      8242      4
: 8134      8243      4
: 8135      8244      4
: 8136      8245      4
: 8137      8246      4
: 8138      8247      4
: 8139      8248      4
: 8140      8249      4
: 8141      8250      4
: 8142      8251      3
: 8143      8252      3
: 8144      8253      2
: 8145      8254      2
: 8146      8255      2
: 8147      8256      2
: 8148      8257      2
: 8149      8258      2
: 8150      8259      2
: 8151      8260      2
: 8152      8261      2
: 8153      8262      1

```

```

! cates a new subnode for the object being pointed to.
[PRIMARY$K_ACT_DEREF_DEREF]:
    GET_DEREFERENCE(.PRIMPTR);

! Handle a PASCAL dereference operator (^) followed by a termi-
! nator. The Primary Descriptor is now complete, so we exit
! the parse loop.
[PRIMARY$K_ACT_DEREF_TERM]:
    EXITLOOP;

! Any other CASE index constitutes an internal DEBUG error.
[INRANGE, OUTRANGE]:
    $DBG_ERROR('DBGPARSER\PRIMARY_PARSER 10');

TES;

END;                                ! End of ELSE-clause for operators

END;                                ! End of the get-token loop

! We are all done parsing the primary. Return a pointer to the
! descriptor we have constructed.
RET TOKEN[0] = .PRIMPTR;
RETURN;

END;

```

```

                                .PSECT DBG$PLIT,NOWRT, SHR, PIC,0
                                4C 30 031A0 TOKEN_IS_INTEGER:
                                .BYTE 48, 76
                                00 031A2 .BLKB 1
                                00 031A3 P.AXG: .BYTE 0
                                00 031A4 P.AXH: .BYTE 0
                                08 031A5 P.AXI: .ASCII <8>\decimal \
                                10 031AE P.AXJ: .ASCII <16>\longword integer\
                                72 65 031BD
                                4D 49 52 50 5C 52 45 53 52 41 50 47 42 44 18 031BF P.AXK: .ASCII <27>\DBGPARSER\<92>\PRIMARY_PARSER 10\
                                30 31 20 52 45 53 52 41 50 5F 59 52 41 031CE

```

```

                                .PSECT DBG$CODE,NOWRT, SHR, PIC,0
                                OFFC 00000
                                5E FBAB CE 9E 00002
                                50 00000000 EF D0 00007
                                .ENTRY DBG$PRIMARY_PARSER, Save R2,R3,R4,R5,R6,R7,-; 6691
                                R8,R9,R10,RT1
                                MOVAB -112(SP), SP
                                MOVL SAVED_TOKEN, R0
                                : 6851

```

			06		6C	91	0000E		CMPB	(AP), #6	6840
					1D	1B	00011		BLEQU	2\$	
					50	D5	00013		TSTL	R0	6851
					06	13	00015		BEQL	1\$	
	14	BC	1C		AC	D0	00017		MOVL	28(AP), @RET_TOKEN	6854
					04	04	0001C		RET		6853
	56				01	CE	0001D	1\$:	MNEGL	#1, LAST_OPERAND	6867
	10	AE	04		AC	D0	00020		MOVL	OPERAND_EXPECTED_FLAG, OPERAND_EXPECTED	6868
	59		20		AC	D0	00025		MOVL	32(AP), STATE_INDEX	6869
	24	AE	1C		AC	D0	00029		MOVL	28(AP), PRIMPTR	6870
					39	11	0002E		BRB	4\$	6840
					50	D5	00030	2\$:	TSTL	R0	6882
					CC	13	00032		BEQL	3\$	
	58				50	D0	00034		MOVL	R0, TOKEN	6885
		00000000'			EF	D4	00037		CLRL	SAVED_TOKEN	6886
				00D1	31	0003D		BRW	9\$		6887
					56	D4	00040	3\$:	CLRL	LAST_OPERAND	6897
	10	AE	04		AC	D0	00042		MOVL	OPERAND_EXPECTED_FLAG, OPERAND_EXPECTED	6898
					59	D4	00047		CLRL	STATE_INDEX	6899
					24	AE	00049		CLRL	PRIMPTR	6900
					08	AE	0004C		CLRL	NUMERIC_PATHNAME	6901
00D0	8F	00	6E		00	2C	0004F		MOVC5	#0, (SP), #0, #208, PATHDESC	6902
				FF30	CD		00056				
			5B	FF38	CD	9E	00059		MOVAB	PATHDESC+8, PATHVECTOR	6903
0270	8F	00	6E		00	2C	0005E		MOVC5	#0, (SP), #0, #624, SUBSCR_DESC	6904
					40	AE	00065				
					57	D4	00067		CLRL	PLIPTR	6905
			7E		0C	AC	00069	4\$:	MOVQ	TERM_LIST, -(SP)	6919
					08	AC	0006D		PUSHL	ADDRESS_EXPRESSION	
					1C	AE	00070		PUSHL	OPERAND_EXPECTED	6918
	F098	CF			04	FB	00073		CALLS	#4, DBG\$LEXICAL_SCANNER	
		58			50	D0	00078		MOVL	R0, TOKEN	
07	00000000G	00			02	E1	0007B		BBC	#2, DBG\$GL_DEVELOPER, 5\$	6920
					58	DD	00083		PUSHL	TOKEN	
	0000V	CF			01	FB	00085		CALLS	#1, DUMP_TOKEN	
		5A	24		AE	D0	0008A	5\$:	MOVL	PRIMPTR, R10	6929
					55	12	0008E		BNEQ	6\$	
		51	10		AE	E8	00090		BLBS	OPERAND_EXPECTED, 6\$	6930
					56	D5	00094		TSTL	LAST_OPERAND	6931
					4D	13	00096		BEQL	6\$	
		01			68	91	00098		CMPB	(TOKEN), #1	6932
					48	12	0009B		BNEQ	6\$	
		50	02		A8	3C	0009D		MOVZWL	2(TOKEN), R0	6933
3C	00000000'	EF			50	E1	000A1		BBC	R0, TOKEN_IS_INTEGER, 6\$	
		50	08		A8	9A	000A9		MOVZBL	8(TOKEN), -R0	6937
		50			04	C0	000AD		ADDL2	#4, R0	
		50			04	C6	000B0		DIVL2	#4, R0	
			03		A0	9F	000B3		PUSHAB	3(R0)	6936
	00000000G	00			01	FB	000B6		CALLS	#1, DBG\$GET_TEMPMEM	
		04			50	D0	000BD		MOVL	R0, TEMPTOKEN	
		04			04	90	000C1		MOVB	#4, @TEMPTOKEN	6938
		04			8F	A8	000C5		BISW2	#256, @TEMPTOKEN	6939
04	BE	10		0100	09	F0	000CB		INSV	#9, #16, #16, @TEMPTOKEN	6940
					08	A8	000D1		MOVZBL	8(TOKEN), R0	6941
					50	D6	000D5		INCL	R0	
	7E	04	AE		0C	C1	000D7		ADDL3	#12, TEMPTOKEN, -(SP)	6942
	9E	08	AB		50	28	000DC		MOVC3	R0, 8(TOKEN), @-(SP)+	

58	04	AE	D0	000E1	68:	MOVL	TEMPTOKEN, TOKEN	6943	
01		68	91	000E5		CMPB	(TOKEN), #1	6951	
		1F	12	000E8		BNEQ	8\$		
12	10	AE	E8	000EA		BLBS	OPERAND_EXPECTED, 7\$	6954	
	08	A8	9F	000EE		PUSHAB	8(TOKEN)	6956	
		01	DD	000F1		PUSHL	#1		
	000289AA	8F	DD	000F3		PUSHL	#166314		
00000000G	00	03	FB	000F9		CALLS	#3, LIB\$SIGNAL		
	10	AE	D4	00100	7\$:	CLRL	OPERAND_EXPECTED	6958	
56		58	D0	00103		MOVL	TOKEN, LAST_OPERAND	6959	
		FF60	31	00106		BRW	4\$	6951	
1E	01	A8	E8	00109	8\$:	BLBS	1(TOKEN), 11\$	6979	
		56	D5	0010D		TSTL	LAST_OPERAND	6986	
		08	12	0010F		BNEQ	10\$		
14	BC	58	D0	00111	9\$:	MOVL	TOKEN, @RET TOKEN	6989	
	18	BC	D4	00115		CLRL	@RET_OPERAND_FLAG	6990	
			04	00118		RET		6988	
00000000G	EF	58	D0	00119	10\$:	MOVL	TOKEN, SAVED_TOKEN	6998	
	58	00000000	EF	9E	00120	MOVAB	PRIMARY_TERM_TOKEN, TOKEN	6999	
18	BC	01	D0	00127		MOVL	#1, @RET_OPERAND_FLAG	7000	
	09	10	AE	E9	0012B	11\$:	BLBC	OPERAND_EXPECTED, 12\$	7010
	02		68	91	0012F		CMPB	(TOKEN), #2	7011
			09	12	00132		BNEQ	13\$	
17	10	AE	E8	00134		BLBS	OPERAND_EXPECTED, 14\$	7012	
02		68	91	00138	12\$:	CMPB	(TOKEN), #2	7013	
		12	12	0013B		BNEQ	14\$		
	0C	A8	9F	0013D	13\$:	PUSHAB	12(TOKEN)	7015	
		01	DD	00140		PUSHL	#1		
	000289B2	8F	DD	00142		PUSHL	#166322		
00000000G	00	03	FB	00148		CALLS	#3, LIB\$SIGNAL		
	02	68	91	0014F	14\$:	CMPB	(TOKEN), #2	7018	
		24	13	00152		BEQL	16\$		
		56	D5	00154		TSTL	LAST_OPERAND	7020	
FFFFFFFF	8F	0E	13	00156		BEQL	15\$		
		56	D1	00158		CMPL	LAST_OPERAND, #-1	7025	
	01	17	13	0015F		BEQL	16\$		
		66	91	00161		CMPB	(LAST_OPERAND), #1	7028	
		12	13	00164		BEQL	16\$		
	0C	A8	9F	00166	15\$:	PUSHAB	12(TOKEN)	7030	
		01	DD	00169		PUSHL	#1		
	000289B2	8F	DD	0016B		PUSHL	#166322		
00000000G	00	03	FB	00171		CALLS	#3, LIB\$SIGNAL		
	04	68	91	00178	16\$:	CMPB	(TOKEN), #4	7035	
		04	13	0017B		BEQL	17\$		
10	AE	01	D0	0017D		MOVL	#1, OPERAND_EXPECTED	7037	
14	AE	02	A8	3C	00181	17\$:	MOVZWL	2(TOKEN), OPCODE	7048
		00000000	FF49	DF	00186	18\$:	PUSHAL	@PRIMARY_TABLE[STATE_INDEX]	7049
14	AE	9E	00	ED	0018D		CMPZV	#0, #8, @SP+, OPCODE	
			21	13	00193		BEQL	20\$	
		00000000	FF49	DF	00195		PUSHAL	@PRIMARY_TABLE[STATE_INDEX]	7051
			9E	95	0019C		TSTB	@SP+	
		12	12	0019E		BNEQ	19\$		
	0C	A8	9F	001A0		PUSHAB	12(TOKEN)	7053	
		01	DD	001A3		PUSHL	#1		
	000289E2	8F	DD	001A5		PUSHL	#166370		
00000000G	00	03	FB	001AB		CALLS	#3, LIB\$SIGNAL		
		59	D6	001B2	19\$:	INCL	STATE_INDEX	7055	

[illegible]

		00028362	01 DD 0023E	PUSHL #1	
00000000G	00		8F DD 00240	PUSHL #164706	
	FF30	CD 0101	03 FB 00246	CALLS #3, LIB\$SIGNAL	
		6B 00000000	76 11 0024D	BRB 26\$	
			8F B0 0024F	MOVW #257, PATHDESC	7081
			EF 9E 00256	MOVAB P.AXG, (PATHVECTOR)	7083
		03BC	31 0025D	BRW 69\$	7084
03 00000000	50	02	A6 3C 00260	MOVZWL 2(LAST_OPERAND), R0	7106
	EF		50 E0 00264	BBS R0, TOKEN_IS_INTEGER, 24\$	
		04CE	31 0026C	BRW 84\$	
	FE5A	CD 010E	8F B0 0026F	MOVW #270, STRDESC+2	7109
	FE58	CD 08	A6 9B 00276	MOVZBW 8(LAST_OPERAND), STRDESC	7111
	FE5C	CD 09	A6 9E 0027C	MOVAB 9(R6), -STRDESC+4	7112
		FF34	CD 9F 00282	PUSHAB PATHDESC+4	7113
		FE58	CD 9F 00286	PUSHAB STRDESC	
00000000G	00		02 FB 0028A	CALLS #2, OT\$SCVT_TI_L	
	OC		50 D0 00291	MOVL R0, STATUS	
		OC	AE E8 00295	BLBS STATUS, 25\$	7114
		OC	AE DD 00299	PUSHL STATUS	7117
		08	A6 9F 0029C	PUSHAB 8(LAST_OPERAND)	
			01 DD 0029F	PUSHL #1	
		0002898A	8F DD 002A1	PUSHL #166282	
00000000G	00		04 FB 002A7	CALLS #4, LIB\$SIGNAL	
	FF30	CD	01 90 002AE	MOVB #1, PATHDESC	7120
	FF31	CD 0101	8F B0 002B3	MOVW #257, PATHDESC+1	7121
		6B 00000000	EF 9E 002BA	MOVAB P.AXH, (PATHVECTOR)	7122
	08	AE	01 D0 002C1	MOVL #1, NUMERIC_PATHNAME	7123
			72 11 002C5	BRB 30\$	7106
		FF32	CD 95 002C7	TSTB PATHDESC+2	7149
			0D 13 002CB	BEQL 28\$	
		00028A12	8F DD 002CD	PUSHL #166418	7151
00000000G	00		01 FB 002D3	CALLS #1, LIB\$SIGNAL	
	FE5A	CD 010E	8F B0 002DA	MOVW #270, STRDESC+2	7157
	FE58	CD OC	A8 9B 002E1	MOVZBW 12(TOKEN), STRDESC	7159
	FE5C	CD OD	A8 9E 002E7	MOVAB 13(R8), STRDESC+4	7160
		FF34	CD 9F 002ED	PUSHAB PATHDESC+4	7161
		FE58	CD 9F 002F1	PUSHAB STRDESC	
00000000G	00		02 FB 002F5	CALLS #2, OT\$SCVT_TI_L	
	OC		50 D0 002FC	MOVL R0, STATUS	
		OC	AE E8 00300	BLBS STATUS, 29\$	7162
		OC	AE DD 00304	PUSHL STATUS	7167
		00000000	EF 9F 00307	PUSHAB P.AXJ	7166
		FE58	CD DD 0030D	PUSHL STRDESC	
		00000000	EF 9F 00311	PUSHAB P.AXI	7165
			03 DD 00317	PUSHL #3	7164
		00028E78	8F DD 00319	PUSHL #167544	
		OC	A8 9F 0031F	PUSHAB 12(TOKEN)	
			01 DD 00322	PUSHL #1	
		0002898A	8F DD 00324	PUSHL #166282	
			09 FB 0032A	CALLS #9, LIB\$SIGNAL	
FF32	CD	00000000G	01 81 00331	ADDB3 #1, PATHDESC, PATHDESC+2	7169
		FF30	CD	BRB 35\$	7064
			46 11 00339	CMPL 2(LAST_OPERAND), #1	7186
		01	A6 B1 0033B	BNEQ 32\$	
			1F 12 0033F	TSTB PATHDESC+2	7194
		FF32	CD 95 00341	BNEQ 32\$	
			19 12 00345	PUSHAB DUMMY	7201
		20	AE 9F 00347		

		24	AE	9F	0034A	PUSHAB	DUMMY		
		2C	AE	9F	0034D	PUSHAB	PRIMPTR		
		3C	AE	9F	00350	PUSHAB	KIND		
00000000G	00	08	A6	9F	00353	PUSHAB	8(LAST_OPERAND)		
	03		05	FB	00356	CALLS	#5, DBG\$DEF_SYM_FIND		
			50	E8	0035D	BLBS	R0, 33\$		
		02	EC	31	00360	BRW	74\$		
	01	30	AE	D1	00363	CMPL	KIND, #1		7206
			06	13	00367	BEQL	34\$		
	05	30	AE	D1	00369	CMPL	KIND, #5		7207
			F1	12	0036D	BNEQ	32\$		
			7E	D4	0036F	CLRL	-(SP)		7216
		24	AE	9F	00371	PUSHAB	DUMMY		
		2C	AE	9F	00374	PUSHAB	PRIMPTR		
		30	AE	DD	00377	PUSHL	PRIMPTR		
00000000G	00		04	FB	0037A	CALLS	#4, DBG\$NCOPY_DESC		
		FC	E5	31	00381	BRW	4\$		7218
	01	02	A6	B1	00384	CMPL	2(LAST_OPERAND), #1		7263
			1F	12	00388	BNEQ	37\$		
		FF	32	CD	95	TSTB	PATHDESC+2		7271
			19	12	0038E	BNEQ	37\$		
		20	AE	9F	00390	PUSHAB	DUMMY		7278
		24	AE	9F	00393	PUSHAB	DUMMY		
		2C	AE	9F	00396	PUSHAB	PRIMPTR		
		3C	AE	9F	00399	PUSHAB	KIND		
		08	A6	9F	0039C	PUSHAB	8(LAST_OPERAND)		
00000000G	00		05	FB	0039F	CALLS	#5, DBG\$DEF_SYM_FIND		
	03		50	E8	003A6	BLBS	R0, 38\$		
		02	C8	31	003A9	BRW	75\$		
		30	AE	D1	003AC	CMPL	KIND, #1		7283
			06	13	003B0	BEQL	39\$		
	05	30	AE	D1	003B2	CMPL	KIND, #5		7284
			F1	12	003B6	BNEQ	37\$		
			7E	D4	003B8	CLRL	-(SP)		7293
		24	AE	9F	003BA	PUSHAB	DUMMY		
		2C	AE	9F	003BD	PUSHAB	PRIMPTR		
		30	AE	DD	003C0	PUSHL	PRIMPTR		
00000000G	00		04	FB	003C3	CALLS	#4, DBG\$NCOPY_DESC		
		02	CC	31	003CA	BRW	76\$		7305
	01	02	A6	B1	003CD	CMPL	2(LAST_OPERAND), #1		7333
			40	12	003D1	BNEQ	42\$		
		FF	32	CD	95	TSTB	PATHDESC+2		7341
			4C	12	003D7	BNEQ	43\$		
		20	AE	9F	003D9	PUSHAB	DUMMY		7348
		24	AE	9F	003DC	PUSHAB	DUMMY		
		2C	AE	9F	003DF	PUSHAB	PRIMPTR		
		3C	AE	9F	003E2	PUSHAB	KIND		
		08	A6	9F	003E5	PUSHAB	8(LAST_OPERAND)		
00000000G	00		05	FB	003E8	CALLS	#5, DBG\$DEF_SYM_FIND		
	33		50	E9	003EF	BLBC	R0, 43\$		
	01	30	AE	D1	003F2	CMPL	KIND, #1		7353
			06	13	003F6	BEQL	41\$		
	05	30	AE	D1	003F8	CMPL	KIND, #5		7354
			27	12	003FC	BNEQ	43\$		
			7E	D4	003FE	CLRL	-(SP)		7363
		24	AE	9F	00400	PUSHAB	DUMMY		
		2C	AE	9F	00403	PUSHAB	PRIMPTR		

00000000G	00	30	AE	DD	00406	PUSHL	PRIMPTR		
			04	FB	00409	CALLS	#4, DBG\$NCOPY_DESC		
		02AF	31	00410	BRW	78\$		7365	
		08	A6	9F	00413	PUSHAB	8(LAST_OPERAND)	7376	
			01	DD	00416	PUSHL	#1		
		000281A8	8F	DD	00418	PUSHL	#164264		
00000000G	00		03	FB	0041E	CALLS	#3, LIB\$SIGNAL		
		0277	31	00425	BRW	77\$		7382	
	01	02	A6	B1	00428	CMPL	2(LAST_OPERAND), #1	7407	
			1F	12	0042C	BNEQ	45\$		
		FF32	CD	95	0042E	TSTB	PATHDESC+2	7415	
			19	12	00432	BNEQ	45\$		
		20	AE	9F	00434	PUSHAB	DUMMY	7422	
		24	AE	9F	00437	PUSHAB	DUMMY		
		2C	AE	9F	0043A	PUSHAB	PRIMPTR		
		3C	AE	9F	0043D	PUSHAB	KIND		
		08	A6	9F	00440	PUSHAB	8(LAST_OPERAND)		
00000000G	00		05	FB	00443	CALLS	#5, DBG\$DEF_SYM_FIND		
	03		50	E8	0044A	BLBS	RO, 46\$		
		027F	31	0044D	BRW	80\$			
	01	30	AE	D1	00450	CMPL	KIND, #1	7427	
			06	13	00454	BEQL	47\$		
	05	30	AE	D1	00456	CMPL	KIND, #5	7428	
			F1	12	0045A	BNEQ	45\$		
			7E	D4	0045C	CLRL	-(SP)	7437	
		24	AE	9F	0045E	PUSHAB	DUMMY		
		2C	AE	9F	00461	PUSHAB	PRIMPTR		
		30	AE	DD	00464	PUSHL	PRIMPTR		
00000000G	00		04	FB	00467	CALLS	#4, DBG\$NCOPY_DESC		
		0281	31	0046E	BRW	81\$		7449	
			7E	D4	00471	CLRL	-(SP)	7465	
			56	DD	00473	PUSHL	LAST_OPERAND		
		FF30	CD	9F	00475	PUSHAB	PATHDESC		
0000V	CF		03	FB	00479	CALLS	#3, APPEND TO PATHNAME		
		FE60	CD	9F	0047E	PUSHAB	SAVED_PATHDESC	7466	
			57	DD	00482	PUSHL	PLIPTR	7467	
		48	AE	9F	00484	PUSHAB	SUBSCR_DESC	7466	
		FF30	CD	9F	00487	PUSHAB	PATHDESC		
0000V	CF		04	FB	0048B	CALLS	#4, PATHNAME_TO_PRIMARY		
	57		50	D0	00490	MOVL	RO, PLIPTR		
	06	07	A7	91	00493	CMPL	7(PLIPTR), #6	7468	
			09	12	00497	BNEQ	49\$		
	10	06	A7	91	00499	CMPL	6(PLIPTR), #16	7469	
			03	12	0049D	BNEQ	49\$		
		03D3	31	0049F	BRW	112\$			
		1C	AE	9F	004A2	PUSHAB	NAME	7474	
		FF30	CD	9F	004A5	PUSHAB	PATHDESC		
00000000G	00		02	FB	004A9	CALLS	#2, DBG\$NPATHDESC_TO_CS		
		1C	AE	DD	004B0	PUSHL	NAME	7475	
		0380	31	004B3	BRW	111\$			
	10	02	A6	B1	004B6	CMPL	2(LAST_OPERAND), #16	7499	
			12	13	004BA	BEQL	51\$		
		0C	A8	9F	004BC	PUSHAB	12(TOKEN)	7501	
			01	DD	004BF	PUSHL	#1		
		000289B2	8F	DD	004C1	PUSHL	#166322		
00000000G	00		03	FB	004C7	CALLS	#3, LIB\$SIGNAL		
		08	A6	9F	004CE	PUSHAB	8(LAST_OPERAND)	7510	

	EBAA	7E	01	A6	9A	004D1	MOVZBL	1(LAST_OPERAND), -(SP)	
		CF		02	FB	004D5	CALLS	#2, DBG\$GET_BIF_ARGUMENTS	
		6E		50	DD	004DA	MOVL	R0, ARG_LIST	
		02	00	BE	D1	004DD	CMPL	2ARG_LIST, #2	7516
				35	14	004E1	BGTR	54\$	
		01	00	BE	D1	004E3	CMPL	2ARG_LIST, #1	7525
		7E		05	12	004E7	BNEQ	52\$	
				68	9A	004E9	MOVZBL	(TOKEN), -(SP)	7529
				02	11	004EC	BRB	53\$	7528
				03	DD	004EE	PUSHL	#3	7532
			08	A6	9F	004F0	PUSHAB	8(LAST_OPERAND)	
		7E	04	A6	9A	004F3	MOVZBL	4(LAST_OPERAND), -(SP)	
	0000V	CF		03	FB	004F7	CALLS	#3, CREATE_OPERATOR_TOKEN	
52		58		50	DD	004FC	MOVL	R0, TOKEN	
		6E		08	C1	004FF	ADDL3	#8, ARG_LIST, R2	7535
				62	DD	00503	PUSHL	(R2)	
53	04	AE		04	C1	00505	ADDL3	#4, ARG_LIST, R3	
				63	DD	0050A	PUSHL	(R3)	
				58	DD	0050C	PUSHL	TOKEN	
	00000000G	00		03	FB	0050E	CALLS	#3, DBG\$EVAL_LANG_OPERATOR	
				02DF	31	00515	BRW	102\$	
			08	A6	9F	00518	PUSHAB	8(LAST_OPERAND)	7546
				01	DD	0051B	PUSHL	#1	
			00028838	8F	DD	0051D	PUSHL	#165944	
				00A7	31	00523	BRW	65\$	
		01	02	A6	B1	00526	CMPL	2(LAST_OPERAND), #1	7563
				03	13	0052A	BEQL	56\$	
				0092	31	0052C	BRW	64\$	
			FF32	CD	95	0052F	TSTB	PATHDESC+2	7571
				28	12	00533	BNEQ	58\$	
		20		AE	9F	00535	PUSHAB	DUMMY	7578
		24		AE	9F	00538	PUSHAB	DUMMY	
		2C		AE	9F	0053B	PUSHAB	PRIMPTR	
		3C		AE	9F	0053E	PUSHAB	KIND	
			08	A6	9F	00541	PUSHAB	8(LAST_OPERAND)	
	00000000G	00		05	FB	00544	CALLS	#5, DBG\$DEF_SYM_FIND	
		0F		50	E9	0054B	BLBC	R0, 58\$	
		01	30	AE	D1	0054E	CMPL	KIND, #1	7583
				04	13	00552	BEQL	57\$	
		05	30	AE	D1	00554	CMPL	KIND, #5	7584
				03	12	00558	BNEQ	58\$	
				00AA	31	0055A	BRW	67\$	
				7E	D4	0055D	CLRL	-(SP)	7602
				56	DD	0055F	PUSHL	LAST_OPERAND	
				FF30	CD	00561	PUSHAB	PATHDESC	
	0000V	CF		03	FB	00565	CALLS	#3, APPEND_TO_PATHNAME	
				01	DD	0056A	PUSHL	#1	7608
				7E	7C	0056C	CLRQ	-(SP)	
				7E	D4	0056E	CLRL	-(SP)	
			40	AE	9F	00570	PUSHAB	KIND	
			48	AE	9F	00573	PUSHAB	TYPEID	
			FF30	CD	9F	00576	PUSHAB	PATHDESC	
	00000000G	00		07	FB	0057A	CALLS	#7, DBG\$STA_GETSYMBOL	
			34	AE	D5	00581	TSTL	TYPEID	7613
				03	13	00584	BEQL	60\$	
				0262	31	00586	BRW	101\$	
			38	AE	9F	00589	PUSHAB	PATHSTRING	7616

00000000G	00	FF30	CD	9F	0058C	PUSHAB	PATHDESC		
	09		02	FB	00590	CALLS	#2, DBG\$NPATHDESC_TO_CS		
		30	AE	D1	00597	CMPL	KIND, #9		7617
			0D	12	00598	BNEQ	61\$		
		38	AE	DD	0059D	PUSHL	PATHSTRING		7619
			01	DD	005A0	PUSHL	#1		
		000281F0	8F	DD	005A2	PUSHL	#164336		
			14	11	005A8	BRB	63\$		
	0D	30	AE	D1	005AA	61\$: CMPL	KIND, #13		7621
			03	13	005AE	BEQL	62\$		
		0226	31	005B0	BRW	99\$			
		38	AE	DD	005B3	62\$: PUSHL	PATHSTRING		7623
			01	DD	005B6	PUSHL	#1		
		000282A8	8F	DD	005B8	PUSHL	#164520		
			0223	31	005BE	63\$: BRW	100\$		7625
	7E	08	A6	9A	005C1	64\$: MOVZBL	8(LAST_OPERAND), -(SP)		7640
			01	DD	005C5	PUSHL	#1		
		000289E2	8F	DD	005C7	PUSHL	#166370		
00000000G	00		03	FB	005CD	65\$: CALLS	#3, LIB\$SIGNAL		
			43	11	005D4	BRB	68\$		7556
	01	02	A6	B1	005D6	66\$: CMPW	2(LAST_OPERAND), #1		7658
			50	12	005DA	BNEQ	71\$		
		FF32	CD	95	005DC	TSTB	PATHDESC+2		7666
			3A	12	005E0	BNEQ	69\$		
		20	AE	9F	005E2	PUSHAB	DUMMY		7673
		24	AE	9F	005E5	PUSHAB	DUMMY		
		2C	AE	9F	005E8	PUSHAB	PRIMPTR		
		3C	AE	9F	005EB	PUSHAB	KIND		
		08	A6	9F	005EE	PUSHAB	8(LAST_OPERAND)		
00000000G	00		05	FB	005F1	CALLS	#5, DBG\$DEF_SYM_FIND		
	21		50	E9	005F8	BLBC	R0, 69\$		
	01	30	AE	D1	005FB	CMPL	KIND, #1		7678
			06	13	005FF	BEQL	67\$		
	05	30	AE	D1	00601	CMPL	KIND, #5		7679
			15	12	00605	BNEQ	69\$		
		7E	D4	00607	67\$: CLRL	-(SP)			7688
		24	AE	9F	00609	PUSHAB	DUMMY		
		2C	AE	9F	0060C	PUSHAB	PRIMPTR		
		30	AE	DD	0060F	PUSHL	PRIMPTR		
00000000G	00		04	FB	00612	CALLS	#4, DBG\$NCPY_DESC		
		02AD	31	00619	68\$: BRW	122\$			7681
		7E	D4	0061C	69\$: CLRL	-(SP)			7697
		56	DD	0061E	70\$: PUSHL	LAST_OPERAND			
		FF30	CD	9F	00620	PUSHAB	PATHDESC		
0000V	CF		03	FB	00624	CALLS	#3, APPEND_TO_PATHNAME		
		0272	31	00629	11\$: BRW	115\$			7698
		56	DD	0062C	71\$: PUSHL	LAST_OPERAND			7707
0000V	CF		01	FB	0062E	CALLS	#1, CONSTANT_TO_VALDESCR		
		027A	31	00633	BRW	116\$			
	12	08	AE	E9	00636	72\$: BLBC	NUMERIC_PATHNAME, 73\$		7724
		08	A6	9F	0063A	PUSHAB	8(LAST_OPERAND)		7726
			01	DD	0063D	PUSHL	#1		
		0002898A	8F	DD	0063F	PUSHL	#166282		
00000000G	00		03	FB	00645	CALLS	#3, LIB\$SIGNAL		
		00EE	31	0064C	73\$: BRW	84\$			7728
		7E	D4	0064F	74\$: CLRL	-(SP)			7738
		56	DD	00651	PUSHL	LAST_OPERAND			

0000V	CF	FF30	CD	9F	00653	PUSHAB	PATHDESC		
			03	FB	00657	CALLS	#3, APPEND TO PATHNAME		
		FE60	CD	9F	0065C	PUSHAB	SAVED PATHDESC		7739
			57	DD	00660	PUSHL	PLIPTR		7740
		48	AE	9F	00662	PUSHAB	SUBSCR DESC		7739
		FF30	CD	9F	00665	PUSHAB	PATHDESC		
0000V	CF		04	FB	00669	CALLS	#4, PATHNAME_TO_PRIMARY		
24	AE		50	DD	0066E	MOVL	R0, PRIMPTR		
			59	11	00672	BRB	79\$		7064
			7E	D4	00674	CLRL	-(SP)		7761
			56	DD	00676	PUSHL	LAST OPERAND		
		FF30	CD	9F	00678	PUSHAB	PATHDESC		
0000V	CF		03	FB	0067C	CALLS	#3, APPEND_TO_PATHNAME		
			01	DD	00681	PUSHL	#1		7762
		FE60	CD	9F	00683	PUSHAB	SAVED PATHDESC		
			57	DD	00687	PUSHL	PLIPTR		7763
		4C	AE	9F	00689	PUSHAB	SUBSCR DESC		7762
		FF30	CD	9F	0068C	PUSHAB	PATHDESC		
0000V	CF		05	FB	00690	CALLS	#5, PATHNAME_TO_PRIMARY		
24	AE		50	DD	00695	MOVL	R0, PRIMPTR		
		24	AE	DD	00699	PUSHL	PRIMPTR		7764
		0219	31	0069C	BRW	118\$			
			7E	D4	0069F	CLRL	-(SP)		7787
			56	DD	006A1	PUSHL	LAST OPERAND		
		FF30	CD	9F	006A3	PUSHAB	PATHDESC		
0000V	CF		03	FB	006A7	CALLS	#3, APPEND TO PATHNAME		
		FE60	CD	9F	006AC	PUSHAB	SAVED PATHDESC		7788
			57	DD	006B0	PUSHL	PLIPTR		7789
		48	AE	9F	006B2	PUSHAB	SUBSCR DESC		7788
		FF30	CD	9F	006B5	PUSHAB	PATHDESC		
0000V	CF		04	FB	006B9	CALLS	#4, PATHNAME_TO_PRIMARY		
24	AE		50	DD	006BE	MOVL	R0, PRIMPTR		
		08	A6	9F	006C2	PUSHAB	8(LAST OPERAND)		7790
		28	AE	DD	006C5	PUSHL	PRIMPTR		
0000V	CF		02	FB	006C8	CALLS	#2, GET_BLISS_SUBSCRIPTS		
			7C	11	006CD	BRB	86\$		7064
			7E	D4	006CF	CLRL	-(SP)		7803
			56	DD	006D1	PUSHL	LAST OPERAND		
		FF30	CD	9F	006D3	PUSHAB	PATHDESC		
0000V	CF		03	FB	006D7	CALLS	#3, APPEND TO PATHNAME		
		FE60	CD	9F	006DC	PUSHAB	SAVED PATHDESC		7804
			57	DD	006E0	PUSHL	PLIPTR		7805
		48	AE	9F	006E2	PUSHAB	SUBSCR DESC		7804
		FF30	CD	9F	006E5	PUSHAB	PATHDESC		
0000V	CF		04	FB	006E9	CALLS	#4, PATHNAME_TO_PRIMARY		
24	AE		50	DD	006EE	MOVL	R0, PRIMPTR		
		24	AE	DD	006F2	PUSHL	PRIMPTR		7806
		01C9	31	006F5	BRW	120\$			
			7E	D4	006F8	CLRL	-(SP)		7822
			56	DD	006FA	PUSHL	LAST OPERAND		
		FF30	CD	9F	006FC	PUSHAB	PATHDESC		
0000V	CF		03	FB	00700	CALLS	#3, APPEND TO PATHNAME		
		FE60	CD	9F	00705	PUSHAB	SAVED PATHDESC		7823
			57	DD	00709	PUSHL	PLIPTR		7824
		48	AE	9F	0070B	PUSHAB	SUBSCR DESC		7823
		FF30	CD	9F	0070E	PUSHAB	PATHDESC		
0000V	CF		04	FB	00712	CALLS	#4, PATHNAME_TO_PRIMARY		

57		50	00	00717	MOVL	R0, PLIPTR		
06	07	A7	91	0071A	CMPB	7(PLIPTR), #6		7825
		09	12	0071E	BNEQ	83\$		
10	06	A7	91	00720	CMPB	6(PLIPTR), #16		7826
		03	12	00724	BNEQ	83\$		
		014C	31	00726	BRW	112\$		
	28	AE	9F	00729	PUSHAB	NAME		7831
	FF30	CD	9F	0072C	PUSHAB	PATHDESC		
00000000G	00	02	FB	00730	CALLS	#2, DBG\$NPATHDESC_TO_CS		
	28	AE	DD	00737	PUSHL	NAME		7832
		0129	31	0073A	BRW	111\$		
		7E	D4	0073D	CLRL	-(SP)		7908
		12	11	0073F	BRB	89\$		
	08	A6	9F	00741	PUSHAB	8(LAST_OPERAND)		7917
		5A	DD	00744	PUSHL	R10		
0000V	CF	02	FB	00746	CALLS	#2, GET_RECORD_COMPONENT		
		11	11	0074B	BRB	90\$		
		01	DD	0074D	PUSHL	#1		7926
		02	11	0074F	BRB	89\$		
		02	DD	00751	PUSHL	#2		7935
		56	DD	00753	PUSHL	LAST_OPERAND		
	FF30	CD	9F	00755	PUSHAB	PATHDESC		
0000V	CF	03	FB	00759	CALLS	#3, APPEND_TO_PATHNAME		
		F908	31	0075E	BRW	4\$		7064
	08	A6	9F	00761	PUSHAB	8(LAST_OPERAND)		7947
		5A	DD	00764	PUSHL	R10		
0000V	CF	02	FB	00766	CALLS	#2, GET_RECORD_COMPONENT		
		0148	31	0076B	BRW	117\$		7948
		01	DD	0076E	PUSHL	#1		7958
		02	11	00770	BRB	94\$		
		7E	D4	00772	CLRL	-(SP)		7968
		56	DD	00774	PUSHL	LAST_OPERAND		
	FF30	CD	9F	00776	PUSHAB	PATHDESC		
0000V	CF	03	FB	0077A	CALLS	#3, APPEND_TO_PATHNAME		
		009A	31	0077F	BRW	107\$		7969
	08	A6	9F	00782	PUSHAB	8(LAST_OPERAND)		7982
		5A	DD	00785	PUSHL	R10		
0000V	CF	02	FB	00787	CALLS	#2, GET_RECORD_COMPONENT		
		0130	31	0078C	BRW	119\$		7983
		01	DD	0078F	PUSHL	#1		7997
		56	DD	00791	PUSHL	LAST_OPERAND		
	FF30	CD	9F	00793	PUSHAB	PATHDESC		
0000V	CF	03	FB	00797	CALLS	#3, APPEND_TO_PATHNAME		
		FE60	CD	0079C	PUSHAB	SAVED_PATHDESC		7998
		57	DD	007A0	PUSHL	PLIPTR		7999
		48	AE	9F	PUSHAB	SUBSCR_DESC		7998
	FF30	CD	9F	007A5	PUSHAB	PATHDESC		
0000V	CF	04	FB	007A9	CALLS	#4, PATHNAME_TO_PRIMARY		
		50	DD	007AE	MOVL	R0, PLIPTR		
57	07	A7	91	007B1	CMPB	7(PLIPTR), #6		8000
		09	12	007B5	BNEQ	97\$		
10	06	A7	91	007B7	CMPB	6(PLIPTR), #16		8001
		03	12	007BB	BNEQ	97\$		
		00B5	31	007BD	BRW	112\$		
	2C	AE	9F	007C0	PUSHAB	NAME		8006
	FF30	CD	9F	007C3	PUSHAB	PATHDESC		
00000000G	00	02	FB	007C7	CALLS	#2, DBG\$NPATHDESC_TO_CS		

		2C	AE	DD	007CE	PUSHL	NAME	8007
		0092	31	DD	007D1	BRW	111\$	
		01	DD	007D4	98\$:	PUSHL	#1	8027
		FD86	31	DD	007D6	BRW	59\$	
		38	AE	DD	007D9	99\$:	PUSHL	PATHSTRING
		01	DD	007DC		PUSHL	#1	8050
		000281F8	8F	DD	007DE	PUSHL	#164344	
00000000G	00	03	FB	007E4	100\$:	CALLS	#3, LIB\$SIGNAL	
		58	DD	007EB	101\$:	PUSHL	TOKEN	8053
		38	AE	DD	007ED	PUSHL	TYPEID	
00000000G	00	02	FB	007F0		CALLS	#2, DBG\$EVAL_ADA_TICK	
24	AE	50	DD	007F7	102\$:	MOVL	R0, PRIMPTR	
18	BC	01	DD	007FB		MOVL	#1, @RET_OPERAND_FLAG	8061
		0A	11	007FF		BRB	104\$	8026
		08	A6	9F	00801	103\$:	PUSHAB	8(LAST_OPERAND)
		5A	DD	00804		PUSHL	R10	8074
0000V	CF	02	FB	00806		CALLS	#2, GET_RECORD_COMPONENT	
		008B	31	0080B	104\$:	BRW	122\$	8073
		01	DD	0080E	105\$:	PUSHL	#1	8084
		FE0B	31	00810		BRW	70\$	8096
		5A	DD	00813	106\$:	PUSHL	R10	8109
0000V	CF	01	FB	00815		CALLS	#1, FIX_UP_PRIMARY	
		77	11	0081A		BRB	113\$	
		40	AE	9F	0081C	107\$:	PUSHAB	SUBSCR_DESC
		FF30	CD	9F	0081F		PUSHAB	PATHDESC
0000V	CF	02	FB	00823		CALLS	#2, SAVE_SUBSCRIPTS	
		69	11	00828		BRB	113\$	7064
		5A	DD	0082A	108\$:	PUSHL	R10	8149
0000V	CF	01	FB	0082C		CALLS	#1, FIX_UP_PRIMARY	
		008B	31	00831		BRW	119\$	8150
		FE60	CD	9F	00834	109\$:	PUSHAB	SAVED_PATHDESC
		57	DD	00838		PUSHL	PLIPTR	8164
		48	AE	9F	0083A		PUSHAB	SUBSCR_DESC
		FF30	CD	9F	0083D		PUSHAB	PATHDESC
0000V	CF	04	FB	00841		CALLS	#4, PATHNAME_TO_PRIMARY	
	57	50	DD	00846		MOVL	R0, PLIPTR	
	06	07	A7	91	00849	CMPB	7(PLIPTR), #6	8166
		06	12	0084D		BNEQ	110\$	
	10	06	A7	91	0084F	CMPB	6(PLIPTR), #16	8167
		20	13	00853		BEQ	112\$	
		3C	AE	9F	00855	110\$:	PUSHAB	NAME
		FF30	CD	9F	00858		PUSHAB	PATHDESC
00000000G	00	02	FB	0085C		CALLS	#2, DBG\$NPATHDESC_TO_CS	
		3C	AE	DD	00863		PUSHL	NAME
		01	DD	00866	111\$:	PUSHL	#1	8173
		00028CA0	8F	DD	00868		PUSHL	#167072
00000000G	00	03	FB	0086E		CALLS	#3, LIB\$SIGNAL	
		24	AE	D4	00875	112\$:	CLRL	PRIMPTR
		08	AE	D4	00878		CLRL	NUMERIC_PATHNAME
0000	8F	00	00	2C	0087B		MOVC5	#0, (SPT, #0, #208, PATHDESC
		FF30	CD		00882			
		FF38	CD	9E	00885	MOVAB	PATHDESC+8, PATHVECTOR	8181
0270	8F	00	00	2C	0088A	MOVC5	#0, (SP), #0, #624, SUBSCR_DESC	8182
		40	AE		00891			
		31	11	00893	113\$:	BRB	121\$	7064
		5A	DD	00895	114\$:	PUSHL	R10	8194
0000V	CF	01	FB	00897		CALLS	#1, FIX_UP_PRIMARY	

			28	11	0089C	BRB	122\$	8193
		FE60	CD	9F	0089E 115\$:	PUSHAB	SAVED PATHDESC	8204
			57	DD	008A2	PUSHL	PLIPTR	8205
		48	AE	9F	008A4	PUSHAB	SUBSCR_DESC	8204
		FF30	CD	9F	008A7	PUSHAB	PATHDESC	
			04	FB	008AB	CALLS	#4, PATHNAME_TO_PRIMARY	
0000V	CF		50	DD	008B0 116\$:	MOVL	R0, PRIMPTR	
24	AE		13	11	008B4	BRB	122\$	8203
			5A	DD	008B6 117\$:	PUSHL	R10	8224
0000V	CF		01	FB	008B8 118\$:	CALLS	#1, GET_SUBSCRIPTS	
			07	11	008BD	BRB	121\$	
			5A	DD	008BF 119\$:	PUSHL	R10	8233
0000V	CF		01	FB	008C1 120\$:	CALLS	#1, GET_DEREFERENCE	
			F7A0	31	008C6 121\$:	BRW	4\$	
14	BC		24	AE	DD 008C9 122\$:	MOVL	PRIMPTR, @RET_TOKEN	8259
			04	008CE	RET			8262

; Routine Size: 2255 bytes, Routine Base: DBG\$CODE + 198E

```
8155 8263 1 ROUTINE APPEND_TO_PATHNAME(PATHDESCR, TOKEN, COMPONENT_KIND): NOVALUE =
8156 8264 1
8157 8265 1 FUNCTION
8158 8266 1 This routine appends a pathname component to an existing Pathname
8159 8267 1 Descriptor. It is called by the Primary Parser during the parsing of
8160 8268 1 the pathname part of Primary Symbols (e.g., A\B\C) to build up the
8161 8269 1 Pathname Descriptor which must eventually be passed to GETSYMBOL to
8162 8270 1 get the symbol's SYMID.
8163 8271 1
8164 8272 1 INPUTS
8165 8273 1 PATHDESCR - A pointer to a Pathname Descriptor to which a new pathname
8166 8274 1 component should be appended.
8167 8275 1
8168 8276 1 TOKEN - A pointer to the Lexical Token Entry for the identifier to
8169 8277 1 be appended to the pathname descriptor.
8170 8278 1
8171 8279 1 COMPONENT_KIND - This tells us what kind of pathname component we
8172 8280 1 are dealing with. It can have one of the following values:
8173 8281 1 NOT_REC_COMP (0) - An ordinary pathname component to be
8174 8282 1 appended onto the end of the pathname,
8175 8283 1 e.g., "A" and "B" would be of this kind
8176 8284 1 in the pathname "A\B.C"
8177 8285 1 REC_COMP (1) - In the above example, "C" comes in as
8178 8286 1 a record component. It is appended to
8179 8287 1 the pathname and the TOTCNT is incremented
8180 8288 1 but not the PTHCNT field.
8181 8289 1 COB_REC_COMP (2) - In COBOL, record components come in
8182 8290 1 first, e.g., "C of B of A", so we have
8183 8291 1 to treat them differently when we are
8184 8292 1 building the pathname.
8185 8293 1
8186 8294 1 OUTPUTS
8187 8295 1 The identifier specified by TOKEN is added to the end of the Pathname
8188 8296 1 Descriptor pointed to by PATHDESCR.
8189 8297 1
8190 8298 1
8191 8299 2 BEGIN
8192 8300 2
8193 8301 2 MAP
8194 8302 2 PATHDESCR: REF PTH$PATHNAME, ! Pointer to Pathname Descriptor
8195 8303 2 TOKEN: REF TOKEN$ENTRY; ! Pointer to Identifier Token Entry
8196 8304 2
8197 8305 2 LOCAL
8198 8306 2 PATHVECTOR: REF VECTOR[,LONG]; ! Pointer to pathname vector in the
8199 8307 2 PATHDESCR Pathname Descriptor
8200 8308 2
8201 8309 2
8202 8310 2
8203 8311 2 ! Make sure we have a valid Identifier Lexical Token Entry. Also make sure
8204 8312 2 it will fit in the Pathname Descriptor.
8205 8313 2
8206 8314 2 IF .TOKEN[TOKEN$W_CODE] NEQ TOKEN$K_IDENTIFIER
8207 8315 2 THEN
8208 8316 2 SIGNAL(DBG$_JLLPATHELEM, 1, TOKEN[TOKEN$B_LENGTH]);
8209 8317 2
8210 8318 2 IF .PATHDESCR[PTH$B_PATHCNT] GEQ DBG$K_MAX_PATHNAME
8211 8319 2 THEN
```

```
.. 8212      8320      SIGNAL(DBG$_PATHTOOLONG, 1, TOKEN[TOKEN$B_LENGTH]);
.. 8213      8321
.. 8214      8322      PATHVECTOR = PATHDESCR[PTH$A_PATHVECTOR];
.. 8215      8323
.. 8216      8324      ! Handle a non-COBOL record component.
.. 8217      8325
.. 8218      8326      !
.. 8219      8327      IF .COMPONENT_KIND EQL REC_COMP
.. 8220      8328      THEN
.. 8221      8329          BEGIN
.. 8222      8330              ! Append the new pathname component to the Pathname Descriptor
.. 8223      8331              ! and then just return.
.. 8224      8332
.. 8225      8333              PATHVECTOR[PATHDESCR[PTH$B_TOTCNT]] = TOKEN[TOKEN$B_LENGTH];
.. 8226      8334              PATHDESCR[PTH$B_TOTCNT] = .PATHDESCR[PTH$B_TOTCNT] + 1;
.. 8227      8335              RETURN;
.. 8228      8336          END;
.. 8229      8337
.. 8230      8338      ! Handle the case where we are adding a record component name in COBOL.
.. 8231      8339      ! Since these come in reverse order, previous ones must be pushed down.
.. 8232      8340
.. 8233      8341      IF .COMPONENT_KIND EQL COB_REC_COMP
.. 8234      8342      THEN
.. 8235      8343          BEGIN
.. 8236      8344              ! Check that we have not seen any pathname components yet.
.. 8237      8345
.. 8238      8346              !
.. 8239      8347              IF .PATHDESCR[PTH$B_PATHCNT] NEQ 0
.. 8240      8348              THEN
.. 8241      8349                  $DBG_ERROR('DBGPARSER\APPEND_TO_PATHNAME 10');
.. 8242      8350
.. 8243      8351              ! Push the previous components.
.. 8244      8352
.. 8245      8353              DECR I FROM .PATHDESCR[PTH$B_TOTCNT] TO 1 DO
.. 8246      8354                  PATHVECTOR[I] = .PATHVECTOR[I-1];
.. 8247      8355
.. 8248      8356              ! Add the new component.
.. 8249      8357
.. 8250      8358              PATHVECTOR[0] = TOKEN[TOKEN$B_LENGTH];
.. 8251      8359
.. 8252      8360              ! Bump the count and we are done.
.. 8253      8361
.. 8254      8362              PATHDESCR[PTH$B_TOTCNT] = .PATHDESCR[PTH$B_TOTCNT] + 1;
.. 8255      8363              RETURN;
.. 8256      8364          END;
.. 8257      8365
.. 8258      8366      ! Finally handle the ordinary pathname components.
.. 8259      8367      ! We want to insert the given component after previous pathname components
.. 8260      8368      ! but before any record components we have already picked up.
.. 8261      8369
.. 8262      8370      IF .COMPONENT_KIND EQL NOT_REC_COMP
.. 8263      8371      THEN
.. 8264      8372          BEGIN
.. 8265      8373
.. 8266      8374              ! We want to insert the name at the PATHCNT position.
.. 8267      8375              ! Push down lower names (record component names).
.. 8268      8376
```

END:

[illegible]

6240	FC	A240	D0	00070	5\$:	BRB	6\$		
F7		50	F5	00076	6\$:	MOVL	-4(PATHVECTOR)[I], (PATHVECTOR)[I]		
62	08	A4	9E	00079		SOBGR	1, 5\$		
		27	11	0007D		MOVAB	8(R4), (PATHVECTOR)		8358
	0C	AC	D5	0007F	7\$:	BRB	10\$		8362
		24	12	00082		TSTL	COMPONENT_KIND		8370
51	01	A3	9A	00084		BNEQ	11\$		
55	01	A1	9E	00088		MOVZBL	1(R3), R1		8377
50		63	9A	0008C		MOVAB	1(R1), R5		
		08	11	0008F		MOVZBL	(R3), 1		8378
6240	FC	A240	D0	00091	8\$:	BRB	9\$		
		50	D7	00097		MOVL	-4(PATHVECTOR)[I], (PATHVECTOR)[I]		
55		50	D1	00099	9\$:	DECL	1		
		F3	18	0009C		CMPL	1, R5		
6241	08	A4	9E	0009E		BGEQ	8\$		
	01	A3	96	000A3		MOVAB	8(R4), (PATHVECTOR)[R1]		8382
		63	96	000A6	10\$:	INCB	1(R3)		8383
		04	000A8	11\$:		INCB	(R3)		8384
						RET			8387

: Routine Size: 169 bytes, Routine Base: DBG\$CODE + 225D

```
8281 8388 1 ROUTINE CHECK_UPSCOPE(SYMID, TYPEID, PRIMPTR, DEPTH) =
8282 8389 1
8283 8390 1 FUNCTION
8284 8391 1     This routine is used to implement "incomplete data qualification"
8285 8392 1     in language BASIC. For example, a user can abbreviate A::B::C with
8286 8393 1     A::C. We are passed in a SYMID for the record component ("C" in
8287 8394 1     the above example), and a TYPEID for the record ("A" in the above
8288 8395 1     example. This routine determines whether we can get to the record
8289 8396 1     by going upscope from the component. If so, it returns TRUE and
8290 8397 1     modifies the Primary to include the intervening component
8291 8398 1     selection.
8292 8399 1
8293 8400 1 INPUTS
8294 8401 1     SYMID - SYMID for the record component.
8295 8402 1     TYPEID - TYPEID for the record.
8296 8403 1     PRIMPTR - Pointer to the Primary being constructed.
8297 8404 1     DEPTH - recursion depth
8298 8405 1
8299 8406 1 OUTPUTS
8300 8407 1     The value TRUE is returned if the component selection is valid.
8301 8408 1     In this case, the Primary is modified to include the intervening
8302 8409 1     component selection.
8303 8410 1     The value FALSE is returned if the component is not a valid
8304 8411 1     component for this record.
8305 8412 1
8306 8413 2 BEGIN
8307 8414 2 MAP
8308 8415 2     SYMID: REF RST$ENTRY,
8309 8416 2     TYPEID: REF RST$ENTRY,
8310 8417 2     PRIMPTR: REF DBG$PRIMARY;
8311 8418 2
8312 8419 2 LOCAL
8313 8420 2     I,
8314 8421 2     COMPSYMID,
8315 8422 2     FCODE,
8316 8423 2     NEW TYPEID,
8317 8424 2     NODEPTR: REF DBG$PRIM NODE,
8318 8425 2     TYP COMPLST: REF VECTOR[.LONG],
8319 8426 2     TYPREFTBL: REF VECTOR[.LONG],
8320 8427 2     U_SYMID: REF RST$ENTRY;
8321 8428 2
8322 8429 2     DBG$GL_CURRENT_PRIMARY = .PRIMPTR;
8323 8430 2
8324 8431 2     ! Check that the RST entry upscope from the given
8325 8432 2     ! record component SYMID is a Type RST Entry.
8326 8433 2     ! Then check whether it matches the given TYPEID for the record.
8327 8434 2
8328 8435 2     U_SYMID = .SYMID[RST$UPSCOPEPTR];
8329 8436 2     IF .U_SYMID[RST$B_KIND] NEQ RST$K_TYPE
8330 8437 2     THEN
8331 8438 2         RETURN FALSE;
8332 8439 2     IF .U_SYMID EQL .TYPEID
8333 8440 2     THEN
8334 8441 2         BEGIN
8335 8442 2
8336 8443 2             ! The TYPEID matches, so the record component SYMID is valid.
8337 8444 2             ! In this case, we do want to perform the component selection.
```

```

8338      8445      |
8339      8446      |
8340      8447      |
8341      8448      |
8342      8449      |
8343      8450      |
8344      8451      |
8345      8452      |
8346      8453      |
8347      8454      |
8348      8455      |
8349      8456      |
8350      8457      |
8351      8458      |
8352      8459      |
8353      8460      |
8354      8461      |
8355      8462      |
8356      8463      |
8357      8464      |
8358      8465      |
8359      8466      |
8360      8467      |
8361      8468      |
8362      8469      |
8363      8470      |
8364      8471      |
8365      8472      |
8366      8473      |
8367      8474      |
8368      8475      |
8369      8476      |
8370      8477      |
8371      8478      |
8372      8479      |
8373      8480      |
8374      8481      |
8375      8482      |
8376      8483      |
8377      8484      |
8378      8485      |
8379      8486      |
8380      8487      |
8381      8488      |
8382      8489      |
8383      8490      |
8384      8491      |
8385      8492      |
8386      8493      |
8387      8494      |
8388      8495      |
8389      8496      |
8390      8497      |
8391      8498      |
8392      8499      |
8393      8500      |
8394      8501      |

! Modify the Primary to include the record component selection.
NODEPTR = .PRIMPTR[DBG$L PRIM BLINK];
NODEPTR[DBG$V PNODE EVAL] = TRUE;
TYPCOMPLST = TYPEID[RST$A_TYPCOMPLST];

! Determine the "component index". This "PNREC_INDEX" field gets
! used in determining the logical successor.
INCR I FROM 0 TO .TYPEID[RST$L_TYPCOMPCNT] - 1 DO
    BEGIN
        COMPSYD = .TYPCOMPLST[I];

        ! If this component is the one we seek, set its index into the Record
        ! Sub-Node and leave the loop.
        IF .SYDID EQL .COMPSYD
        THEN
            BEGIN
                NODEPTR[DBG$W_PNREC_INDEX] = .I + 1;
                EXITLOOP;
            END;
        END;

        ! Check for FCODE of record - otherwise, it is not valid to skip
        ! this component.
        DBG$STA SYMTYPE(.SYDID, FCODE, NEW_TYPEID);
        IF .FCODE NEQ RST$K_TYPE_RECORD
        THEN
            RETURN FALSE;

        ! Finally append another Primary Descriptor Sub-Node for the selected
        ! record component. Then return.
        DBG$BUILD_PRIMARY_SUBNODE(.PRIMPTR, RST$K_TYPCOMP, .SYDID,
                                .FCODE, .NEW_TYPEID, 0);
        RETURN TRUE;
    END;

! The immediate upscope pointer did not match. In this case, go further
! upscope. We do this by getting the Type Reference table for this
! Type RST entry. For each SYDID in the table, we recursively call
! the CHECK_UPSCOPE routine.
TYPREFBL = .U_SYDID[RST$L_TYPREFTBL];
I = 0;
WHILE .TYPREFBL[I] NEQ 0 DO
    BEGIN
        U_SYDID = .TYPREFBL[I];
        IF CHECK_UPSCOPE(.U_SYDID, .TYPEID, .PRIMPTR, .DEPTH+1)
        THEN
            BEGIN
                ! If we are at the top level of recursion, then do not modify
                ! the Primary here. This is because it will get done in

```

```

8395      GET_RECORD_COMPONENT.
8396
8397      IF .DEPTH EQL 0
8398      THEN
8399          RETURN TRUE;
8400
8401      ! The TYPEID matches, so the record component SYMID is valid.
8402      ! In this case, we do want to perform the component selection.
8403      ! Modify the Primary to include the record component selection.
8404
8405      NODEPTR = .PRIMPTR[DBG$L PRIM BLINK];
8406      NODEPTR[DBG$V_PNODE_EVAL] = TRUE;
8407      TYPCOMPLST = TYPEID[RST$A_TYPCOMPLST];
8408
8409      ! Determine the 'component index'. This 'PNREC_INDEX' field gets
8410      ! used in determining the logical successor.
8411
8412      INCR I FROM 0 TO .TYPEID[RST$L_TYPCOMPNT] - 1 DO
8413      BEGIN
8414          COMPSYMID = .TYPCOMPLST[I];
8415
8416          ! If this component is the one we seek, set its index into the Record
8417          ! Sub-Node and leave the loop.
8418
8419          IF .SYMID EQL .COMPSYMID
8420          THEN
8421              BEGIN
8422                  NODEPTR[DBG$W_PNREC_INDEX] = .I + 1;
8423                  EXITLOOP;
8424                  END;
8425              END;
8426
8427      ! Check for FCODE of record - otherwise, it is not valid to skip
8428      ! this component.
8429
8430      DBG$STA SYMTYPE(.SYMID, FCODE, NEW_TYPEID);
8431      IF .FCODE NEQ RST$K_TYPE_RECORD
8432      THEN
8433          RETURN FALSE;
8434
8435      ! Finally append another Primary Descriptor Sub-Node for the selected
8436      ! record component. Then return.
8437
8438      DBG$BUILD_PRIMARY_SUBNODE(.PRIMPTR, RST$K_TYPCOMP, .SYMID,
8439                                .FCODE, .NEW_TYPEID, 0);
8440
8441      RETURN TRUE;
8442      END;
8443
8444      I = .I + 1;
8445      END;
8446
8447      ! We failed to find a path to the desired TYPEID. Return FALSE.
8448
8449      RETURN FALSE;
8450      END;
8451
```


				OFFC 00000 CHECK_UPSCOPE:					
		5E	08	C2	00002	.WORD	Save R2,R3,R4,R5,R6,R7,R8,R9,R10,R11	8388	
		56	0C	AC	D0	00005	SUBL2	#8, SP	
00000000G		00		56	D0	00009	MOVL	PRIMPTR, R6	8429
		58	04	AC	D0	00010	MOVL	R6, DBG\$GL_CURRENT_PRIMARY	8435
		57	10	A8	D0	00014	MOVL	SYMID, R8	
		07	14	A7	91	00018	MOVL	16(R8), U_SYMID	
				03	13	0001C	CMPB	20(U_SYMID), #7	8436
				00A6	31	0001E	BEQL	1\$	
		55	08	AC	D0	00021	BRW	12\$	
		55		57	D1	00025	MOVL	TYPEID, R5	8439
				21	12	00028	CPL	U_SYMID, R5	
		53	18	A6	D0	0002A	BNEQ	4\$	
0A		A3		01	88	0002E	MOVL	24(R6), NODEPTR	8447
		52	2C	A5	9E	00032	BISB2	#1, 10(NODEPTR)	8448
		50		01	CE	00036	MOVAB	44(R5), TYPCOMPLST	8449
				09	11	00039	MNEGL	#1, I	8456
		59		6240	D0	0003B	BRB	3\$	
		59		58	D1	0003F	MOVL	(TYPCOMPLST)[I], COMPSYD	
				48	13	00042	CPL	R8, COMPSYD	8462
F2		50	28	A5	F2	00044	BEQL	7\$	
				4D	11	00049	AOBLSS	40(R5), I, 2\$	8454
		5A	1C	A7	D0	0004B	BRB	9\$	8473
				54	D4	0004F	MOVL	28(U_SYMID), TYPREFTBL	8491
5B	10	AC		01	C1	00051	CLRL	I	8492
				6A44	D5	00056	ADDL3	#1, DEPTH, R11	8496
				6C	13	00059	TSTL	(TYPREFTBL)[I]	8493
		57		6A44	D0	0005B	BEQL	12\$	
				58	DD	0005F	MOVL	(TYPREFTBL)[I], U_SYMID	8495
		7E		55	7D	00061	PUSHL	R11	8496
				57	DD	00064	MOVQ	R5, -(SP)	
		96		04	FB	00066	PUSHL	U_SYMID	
		56		50	E9	0006A	CALLS	#7, CHECK_UPSCOPE	
			10	AC	D5	0006D	BLBC	R0, 11\$	
				4D	13	00070	TSTL	DEPTH	8504
		53	18	A6	D0	00072	BEQL	10\$	
0A		A3		01	88	00076	MOVL	24(R6), NODEPTR	8512
		52		01	88	00076	BISB2	#1, 10(NODEPTR)	8513
		50	2C	A5	9E	0007A	MOVAB	44(R5), TYPCOMPLST	8514
				01	CE	0007E	MNEGL	#1, I	8521
				10	11	00081	BRB	8\$	
		59		6240	D0	00083	MOVL	(TYPCOMPLST)[I], COMPSYD	
		59		58	D1	00087	CPL	R8, COMPSYD	8527
				07	12	0008A	BNEQ	8\$	
18	A3	50		01	A1	0008C	ADDW3	#1, I, 24(NODEPTR)	8530
				05	11	00091	BRB	9\$	8529
	EB	50	28	A5	F2	00093	AOBLSS	40(R5), I, 6\$	8519
				5E	DD	00098	PUSHL	SP	8538
			08	AE	9F	0009A	PUSHAB	FCODE	
				58	DD	0009D	PUSHL	R8	
00000000G		00		03	FB	0009F	CALLS	#3, DBG\$STA_SYMTYPE	
		07	04	AE	D1	000A6	CPL	FCODE, #7	8539

		1B	12	000AA		BNEQ	128		
		7E	D4	000AC		CLRL	-(SP)		8547
	04	AE	DD	000AE		PUSHL	NEW TYPEID		8548
	0C	AE	DD	000B1		PUSHL	FCODE		
		58	DD	000B4		PUSHL	R8		8547
		0A	DD	000B6		PUSHL	#10		
		56	DD	000B8		PUSHL	R6		
DCA7	CF	06	FB	000BA		CALLS	#6, DBG\$BUILD_PRIMARY_SUBNODE		
	50	01	D0	000BF	10\$:	MOVL	#1, R0		8549
			04	000C2		RET			
		54	D6	000C3	11\$:	INCL	1		8552
		8F	11	000C5		BRB	5\$		8493
		50	D4	000C7	12\$:	CLRL	R0		8558
			04	000C9		RET			

; Routine Size: 202 bytes, Routine Base: DBG\$CODE + 2306

```
8453 8559 1 ROUTINE CONSTANT_TO_VALDESCR(TOKEN) =
8454 8560 1
8455 8561 1 FUNCTION
8456 8562 1     This routine converts a Constant Lexical Token Entry into the corre-
8457 8563 1     sponding Value Descriptor. For string and character constants, this
8458 8564 1     results in an ordinary Value Descriptor, but for numeric constants
8459 8565 1     it results in an "unconverted" Value Descriptor, i.e., a descriptor
8460 8566 1     in which the numeric constant is represented as the original input
8461 8567 1     character string. The actual conversion of a numeric constant to its
8462 8568 1     binary representation is thus delayed until the appropriate precision
8463 8569 1     of the binary representation can be determined from context.
8464 8570 1
8465 8571 1 INPUTS
8466 8572 1     TOKEN - A pointer to a Constant Lexical Token Entry. This entry
8467 8573 1     represents a numeric, string, or character constant of
8468 8574 1     some sort.
8469 8575 1
8470 8576 1 OUTPUTS
8471 8577 1     A Value Descriptor is constructed for the constant and a pointer to
8472 8578 1     that descriptor is returned as the routine value.
8473 8579 1
8474 8580 1
8475 8581 2 BEGIN
8476 8582 2
8477 8583 2 MAP
8478 8584 2     TOKEN: REF TOKEN$ENTRY;           ! Pointer to Lexical Token Entry
8479 8585 2
8480 8586 2 LOCAL
8481 8587 2     VALPTR: REF DBG$VALDESC;           ! Pointer to Value Descriptor we build
8482 8588 2
8483 8589 2
8484 8590 2
8485 8591 2     ! Build a skeleton Value Descriptor for the constant and copy the constant
8486 8592 2     ! character representation into that descriptor.
8487 8593 2
8488 8594 2     VALPTR = DBG$MAKE_SKELETON_DESC(DBG$K_VALUE_DESC, .TOKEN[TOKEN$B_LENGTH]);
8489 8595 2     VALPTR[DBG$B_DHDR_LANG] = .DBG$GB_LANGUAGE;
8490 8596 2     VALPTR[DBG$B_DHDR_KIND] = RST$K_DATA;
8491 8597 2     VALPTR[DBG$B_DHDR_FCODE] = RST$K_TYPE_ATOMIC;
8492 8598 2     VALPTR[DBG$V_DHDR_UNCVT] = TRUE;
8493 8599 2     VALPTR[DBG$B_VALUE_CLASS] = DSC$K_CLASS_S;
8494 8600 2     VALPTR[DBG$W_VALUE_LENGTH] = .TOKEN[TOKEN$B_LENGTH];
8495 8601 2     VALPTR[DBG$L_VALUE_POINTER] = VALPTR[DBG$A_VALUE_ADDRESS];
8496 8602 2     VALPTR[DBG$W_VALUE_TOKENCODE] = .TOKEN[TOKEN$W_CODE];
8497 8603 2     CH$MOVE(.TOKEN[TOKEN$B_LENGTH], TOKEN[TOKEN$A_NAME],
8498 8604 2             VALPTR[DBG$A_VALUE_ADDRESS]);
8499 8605 2
8500 8606 2
8501 8607 2     ! Determine what kind of constant this is and set the data type and the
8502 8608 2     ! "unconverted" bit accordingly.
8503 8609 2
8504 8610 2 CASE .TOKEN[TOKEN$W_CODE] FROM TOKEN$K_MIN_OPERAND TO TOKEN$K_MAX_OPERAND OF
8505 8611 2     SET
8506 8612 2
8507 8613 2     ! Handle character string constants. Mark the data as type T.
8508 8614 2
8509 8615 2
```

```
8510 8616
8511 8617
8512 8618
8513 8619
8514 8620
8515 8621
8516 8622
8517 8623
8518 8624
8519 8625
8520 8626
8521 8627
8522 8628
8523 8629
8524 8630
8525 8631
8526 8632
8527 8633
8528 8634
8529 8635
8530 8636
8531 8637
8532 8638
8533 8639
8534 8640
8535 8641
8536 8642
8537 8643
8538 8644
8539 8645
8540 8646
8541 8647
8542 8648
8543 8649
8544 8650
8545 8651
8546 8652
8547 8653
8548 8654
8549 8655
8550 8656
8551 8657
8552 8658
8553 8659
8554 8660
8555 8661
8556 8662
8557 8663
8558 8664
8559 8665
8560 8666
8561 8667
8562 8668
8563 8669
8564 8670
8565 8671
8566 8672

[TOKEN$K_STRING]:
  VALPTR[DBG$B_VALUE_DTYPE] = DSC$K_DTYPE_T;

! Handle bit string constants. Mark the data as type TF.
[TOKEN$K_BIT_STRING]:
  VALPTR[DBG$B_VALUE_DTYPE] = DSC$K_DTYPE_V;

! Handle decimal integer constants. Mark the data as type L.
[TOKEN$K_INTEGER]:
  VALPTR[DBG$B_VALUE_DTYPE] = DSC$K_DTYPE_L;

! Handle hexadecimal integer constants. Mark the data as type L.
[TOKEN$K_HEX_INTEGER]:
  VALPTR[DBG$B_VALUE_DTYPE] = DSC$K_DTYPE_L;

! Handle octal integer constants. Mark the data as type L.
[TOKEN$K_OCT_INTEGER]:
  VALPTR[DBG$B_VALUE_DTYPE] = DSC$K_DTYPE_L;

! Handle binary integer constants. Mark the data as type L.
[TOKEN$K_BIN_INTEGER]:
  VALPTR[DBG$B_VALUE_DTYPE] = DSC$K_DTYPE_L;

! Handle floating-point constants without exponents. Mark the data
! type as F.
[TOKEN$K_FLOATING]:
  VALPTR[DBG$B_VALUE_DTYPE] = DSC$K_DTYPE_F;

! Handle floating-point constants with E exponents. Mark the data
! type as F.
[TOKEN$K_EXP_E_FLOAT]:
  VALPTR[DBG$B_VALUE_DTYPE] = DSC$K_DTYPE_F;

! Handle floating-point constants with D exponents. Mark the data
! type as D.
[TOKEN$K_EXP_D_FLOAT]:
  IF .DBG$B_MOD_PTR[MODE_G_FLOATS]
  THEN
    BEGIN
      VALPTR[DBG$B_VALUE_DTYPE] = DSC$K_DTYPE_G;
      VALPTR[DBG$B_VALUE_TOKENCODE] = TOKEN$K_EXP_G_FLOAT;
```



```
8567      END
8568
8569      ELSE
8570          VALPTR[DBG$B_VALUE_DTYPE] = DSC$K_DTYPE_D;
8571
8572      ! Handle floating-point constants with G exponents. Mark the data
8573      ! type as G.
8574      [TOKEN$K_EXP G FLOAT]:
8575          VALPTR[DBG$B_VALUE_DTYPE] = DSC$K_DTYPE_G;
8576
8577      ! Handle floating-point constants with Q exponents. Mark the data
8578      ! type as H.
8579      [TOKEN$K_EXP Q FLOAT]:
8580          VALPTR[DBG$B_VALUE_DTYPE] = DSC$K_DTYPE_H;
8581
8582      ! Handle pack decimal constants. Mark the data type as P.
8583      [TOKEN$K_PACK DECIMAL]:
8584          VALPTR[DBG$B_VALUE_DTYPE] = DSC$K_DTYPE_P;
8585
8586      ! Any other case is an internal error.
8587      [INRANGE, OVRANGE]:
8588          $DBG_ERROR('DBGPARSER\CONSTANT_TO_VALDESC');
8589
8590      TES:
8591
8592      ! The Value Descriptor is constructed. Return its address to the caller.
8593      RETURN .VALPTR;
8594
8595      END;
```

```
53 4E 4F 43 5C 52 45 53 52 41 50 47 42 44 1D 031FB P.AXM: .PSECT DBG$PLIT,NOWRT, SHR, PIC,0
53 53 45 44 4C 41 56 5F 4F 54 5F 54 4E 41 54 0320A .ASCII <29>\DBGPARSER\<92>\CONSTANT_TO_VALDES\
43 03218 .ASCII \C\
```

```
.PSECT DBG$CODE,NOWRT, SHR, PIC,0
01FC 0000 CONSTANT_TO_VALDESC:
58 04 AC D0 00002 .WORD Save R2,R3,R4,R5,R6,R7,R8
7E 08 A8 9A 00006 MOVL TOKEN, R8
7E 7A 8F 9A 0000A MOVZBL 8(R8), -(SP)
MOVZBL #122, -(SP)
```

```
: 8559
: 8594
:
```

00000000G	00		02	FB	0000E	CALLS	#2, DBG\$MAKE_SKELETON_DESC	
	56		50	DD	00015	MOVL	R0, VALPTR	8595
03	A6	00000000G	00	90	00018	MOVB	DBG\$GB_LANGUAGE, 3(VALPTR)	8597
06	A6	0602	8F	B0	00020	MOVW	#1538, -6(VALPTR)	8598
04	A6		20	88	00026	BISB2	#32, 4(VALPTR)	8599
	57	14	A6	9E	0002A	MOVAB	20(VALPTR), R7	
03	A7		01	90	0002E	MOVB	#1, 3(R7)	8600
	67	08	A8	9B	00032	MOVZBW	8(R8), (R7)	8601
18	A6	20	A6	9E	00036	MOVAB	32(VALPTR), 24(VALPTR)	8602
10	A6	02	A8	B0	0003B	MOVW	2(R8), 16(VALPTR)	8603
	50	08	A8	9A	00040	MOVZBL	8(R8), R0	8604
20	A6		50	2B	00044	MOVW	R0, 9(R8), 32(VALPTR)	8610
	OF	02	A8	AF	0004A	MOVW	2(R8), #1, #15	
0043	0020	0037	0020		0004F	CASEW	2(R8), #1, #15	
004F	0049	0049	0043	1\$:	00057	.WORD	2\$-1\$,-	
0020	0043	0043	0070		0005F		3\$-1\$,-	
0020	006A	0076	003D		00067		2\$-1\$,-	
							5\$-1\$,-	
							5\$-1\$,-	
							6\$-1\$,-	
							6\$-1\$,-	
							7\$-1\$,-	
							10\$-1\$,-	
							5\$-1\$,-	
							5\$-1\$,-	
							2\$-1\$,-	
							4\$-1\$,-	
							11\$-1\$,-	
							9\$-1\$,-	
							2\$-1\$	
		00000000'	EF	9F	0006F	PUSHAB	P.AXM	8702
			01	DD	00075	PUSHL	#1	
		00028362	8F	DD	00077	PUSHL	#164706	
00000000G	00		03	FB	0007D	CALLS	#3, LIB\$SIGNAL	
			43	11	00084	BRB	12\$	
02	A7		0E	90	00086	MOVB	#14, 2(R7)	8617
			3D	11	0008A	BRB	12\$	
02	A7		01	90	0008C	MOVB	#1, 2(R7)	8623
			37	11	00090	BRB	12\$	
02	A7		08	90	00092	MOVB	#8, 2(R7)	8647
			31	11	00096	BRB	12\$	
02	A7		0A	90	00098	MOVB	#10, 2(R7)	8661
			2B	11	0009C	BRB	12\$	
	50	00000000G	00	DD	0009E	MOVL	DBG\$GB_MOD_PTR, R0	8668
	0A	09	A0	E9	000A5	BLBC	9(R0), -8\$	
02	A7		1B	90	000A9	MOVB	#27, 2(R7)	8671
10	A6		0F	B0	000AD	MOVW	#15, 16(VALPTR)	8672
			16	11	000B1	BRB	12\$	8668
02	A7		0B	90	000B3	MOVB	#11, 2(R7)	8676
			10	11	000B7	BRB	12\$	8668
02	A7		1B	90	000B9	MOVB	#27, 2(R7)	8683
			0A	11	000BD	BRB	12\$	
02	A7		1C	90	000BF	MOVB	#28, 2(R7)	8690
			04	11	000C3	BRB	12\$	
02	A7		15	90	000C5	MOVB	#21, 2(R7)	8696
	50		56	DD	000C9	MOVL	VALPTR, R0	8709
			04		000CC	RET		8711

DBGPARSER
V04-000

N 2
16-Sep-1984 02:10:13
14-Sep-1984 12:17:30

VAX-11 Bliss-32 V4.0-742
[DEBUG.SRC]DBGPARSER.B32;1

Page 274
(28)

: Routine Size: 205 bytes, Routine Base: DBG\$CODE + 23D0

```

8607 8712 1 ROUTINE CONVERT_TO_INTEGER (VALPTR) =
8608 8713 1
8609 8714 1 FUNCTION
8610 8715 1     This routine converts a value descriptor to an integer value
8611 8716 1     and returns the integer value.
8612 8717 1
8613 8718 1 INPUTS
8614 8719 1     VALPTR -           A pointer to a value descriptor
8615 8720 1
8616 8721 1 OUTPUTS
8617 8722 1     An integer value is returned.
8618 8723 1
8619 8724 1 BEGIN
8620 8725 1
8621 8726 1 LOCAL
8622 8727 1     NEW_VALPTR: REF DBG$VALDESC;      ! New value descriptor
8623 8728 1
8624 8729 1     ! Build a new value descriptor of type longword integer.
8625 8730 1     !
8626 8731 1     NEW_VALPTR = DBG$MAKE_SKELETON_DESC(DBG$K_VALUE_DESC, 4);
8627 8732 1     NEW_VALPTR[DBG$B_DHDR_KIND] = RST$K_DATA;
8628 8733 1     NEW_VALPTR[DBG$B_DHDR_FCODE] = RST$K_TYPE_ATOMIC;
8629 8734 1     NEW_VALPTR[DBG$B_VALUE_CLASS] = DSC$K_CLASS_S;
8630 8735 1     NEW_VALPTR[DBG$B_VALUE_DTYPE] = DSC$K_DTYPE_L;
8631 8736 1     NEW_VALPTR[DBG$W_VALUE_LENGTH] = 4;
8632 8737 1     NEW_VALPTR[DBG$L_VALUE_POINTER] = NEW_VALPTR[DBG$A_VALUE_ADDRESS];
8633 8738 1
8634 8739 1
8635 8740 1     ! Call the routine which does type conversion.
8636 8741 1     !
8637 8742 1     NEW_VALPTR = DBG$EVAL LANG_OPERATOR (
8638 8743 1         DBG$GL_CONVERT_TOKEN, .VALPTR, .NEW_VALPTR);
8639 8744 1
8640 8745 1     ! Return the value in the new value descriptor.
8641 8746 1     !
8642 8747 1     RETURN .NEW_VALPTR [DBG$L_VALUE_VALUE0];
8643 8748 1
8644 8749 1 END;

```

0000 0000 CONVERT_TO_INTEGER:						
			04 DD 00002	WORD	Save nothing	8712
			8F 9A 00004	PUSHL	#4	8732
	7E	7A	02 FB 00008	MOVZBL	#122, -(SP)	
00000000G	00		8F B0 0000F	CALLS	#2, DBG\$MAKE_SKELETON_DESC	8734
06 A0	0602		A0 9E 0001D	MOVW	#1538, 6(NEW_VALPTR)	8737
14 A0	01080004		50 DD 00022	MOVL	#17301508, 20(NEW_VALPTR)	8738
18 A0	20		AC DD 00024	MOVAB	32(R0), 24(NEW_VALPTR)	8744
		04	EF 9F 00027	PUSHL	NEW_VALPTR	
		00000000'	03 FB 0002D	PUSHL	VALPTR	8743
00000000G	00		A0 D0 00034	PUSHAB	DBG\$GL_CONVERT_TOKEN	
50	20		04 00038	CALLS	#3, DBG\$EVAL LANG_OPERATOR	8748
				MOVL	32(NEW_VALPTR), R0	8749
				RET		

DBGPARSER
V04-000

C 3
16-Sep-1984 02:10:13
14-Sep-1984 12:17:30

VAX-11 Bliss-32 V4.0-742
[DEBUG.SRC]DBGPARSER.B32;1

Page 276
(29)

: Routine Size: 57 bytes, Routine Base: DBG\$CODE + 249D

```

8646 8750 1 ROUTINE CREATE_OPERAND_TOKEN(TOKEN_CODE, TOKENBUFFER) =
8647 8751 1
8648 8752 1 FUNCTION
8649 8753 1 This routine creates a Lexical Token Entry for an operand and returns
8650 8754 1 a pointer to the created Token Entry. The Token Entry is created in
8651 8755 1 temporary memory. When returned, the Lexical Token Entry will have
8652 8756 1 the TOKEN$B_KIND field set to TOKEN$K_OPERAND and the TOKEN$W_CODE
8653 8757 1 field set the specified operand token code. The token name or charac-
8654 8758 1 ter representation will also be filled into the Token Entry in Counted
8655 8759 1 ASCII.
8656 8760 1
8657 8761 1 INPUTS
8658 8762 1 TOKEN_CODE - This is the code value which indicates which kind of
8659 8763 1 operand this is. (TOKEN$K_IDENTIFIER or TOKEN$K_STRING
8660 8764 1 would be valid examples.) This value is filled into the
8661 8765 1 TOKEN$W_CODE field.
8662 8766 1
8663 8767 1 TOKENBUFFER - A pointer to a buffer which contains the Counted ASCII
8664 8768 1 representation of the operand for which a Token Entry is to
8665 8769 1 be created. This buffer will thus contain an identifier
8666 8770 1 name, the character representation of a numeric constant,
8667 8771 1 or the contents of a character string constant, depending
8668 8772 1 on the kind of operand involved.
8669 8773 1
8670 8774 1 OUTPUTS
8671 8775 1 A pointer to a Lexical Token Entry for the specified operand is
8672 8776 1 returned as the routine value.
8673 8777 1
8674 8778 1
8675 8779 2 BEGIN
8676 8780 2
8677 8781 2 MAP
8678 8782 2 TOKENBUFFER: REF VECTOR[,BYTE]; ! Pointer to token Counted ASCII string
8679 8783 2
8680 8784 2 LOCAL
8681 8785 2 TOKEN: REF TOKEN$ENTRY; ! Pointer to new Token Entry created
8682 8786 2
8683 8787 2
8684 8788 2
8685 8789 2 ! Get a temporary memory block for the Operand Lexical Token Entry.
8686 8790 2 ! Fill in the fields of the Token Entry and return its address.
8687 8791 2
8688 8792 2 TOKEN = DBG$GET_TEMPMEM(TOKEN$K_ENTSIZE + .TOKENBUFFER[0]/%UPVAL + 1);
8689 8793 2 TOKEN[TOKEN$B_KIND] = TOKEN$K_OPERAND;
8690 8794 2 TOKEN[TOKEN$W_CODE] = .TOKEN_CODE;
8691 8795 2 CH$MOVE(.TOKENBUFFER[0] + 1, .TOKENBUFFER[0], TOKEN[TOKEN$B_LENGTH]);
8692 8796 2 RETURN .TOKEN;
8693 8797 2
8694 8798 1 END;
```

007C 00000 CREATE_OPERAND_TOKEN:

50

08 BC 9A 00002

WORD Save R2,R3,R4,R5,R6
MOVZBL @TOKENBUFFER, R0: 8750
: 8792

DBGPARSER
V04-000

E 3
16-Sep-1984 02:10:13
14-Sep-1984 12:17:30

VAX-11 Bliss-32 V4.0-742
[DEBUG.SRC]DBGPARSER.B32;1

Page 278
(30)

		50		04	C6	00006	DIVL2	#4, R0
			03	A0	9F	00009	PUSHAB	3(R0)
	00000000G	00		01	FB	0000C	CALLS	#1, DBG\$GET_TEMP MEM
		56		50	D0	00013	MOVL	R0, TOKEN
		66		01	90	00016	MOVB	#1, (TOKEN)
	02	A6	04	AC	B0	00019	MOVW	TOKEN CODE, 2(TOKEN)
		50	08	BC	9A	0001E	MOVZBL	@TOKENBUFFER, R0
				50	D6	00022	INCL	R0
08	A6	08		50	28	00024	MOVC3	R0, @TOKENBUFFER, 8(TOKEN)
		50		56	D0	0002A	MOVL	TOKEN, R0
				04	00	0002D	RET	

.....
8793
8794
8795
.....
8796
8798

; Routine Size: 46 bytes, Routine Base: DBG\$CODE + 24D6

```
8696 8799 1 ROUTINE CREATE_OPERATOR_TOKEN(TOKEN_CODE, TOKENBUFFER, KIND) =
8697 8800 1
8698 8801 1 FUNCTION
8699 8802 1     This routine creates a Token Entry for an operator and returns a
8700 8803 1     pointer to the created Token Entry. The Token Entry is created in
8701 8804 1     temporary memory. When returned, the Token Entry will have the
8702 8805 1     TOKENSB_KIND field set to the specified kind and the TOKENSW_CODE
8703 8806 1     field set the specified operator token code. The token name or
8704 8807 1     character representation will also be filled into the Token Entry
8705 8808 1     in Counted ASCII.
8706 8809 1
8707 8810 1 INPUTS
8708 8811 1     TOKEN_CODE - This is the code value which indicates which type of
8709 8812 1     operator this is. (TOKENSK_DOT, TOKENSK_ADD, TOKENSK_BIF_OP
8710 8813 1     would be valid examples.) This value is filled into the
8711 8814 1     TOKENSW_CODE field.
8712 8815 1
8713 8816 1     TOKENBUFFER - A pointer to a buffer which contains the Counted ASCII
8714 8817 1     representation of the operator for which a Token Entry is to
8715 8818 1     be created. This buffer will thus contain an identifier
8716 8819 1     name, or the contents of a character string constant,
8717 8820 1     depending on the kind of operator involved.
8718 8821 1
8719 8822 1     KIND - This is the code value which indicates which kind of operator
8720 8823 1     this is. (e.g. TOKENSK_PREFIX_OP) This value is filled
8721 8824 1     into the TOKENSB_KIND field.
8722 8825 1
8723 8826 1 OUTPUTS
8724 8827 1     A pointer to a Token Entry for the specified operand is returned as
8725 8828 1     the routine value.
8726 8829 1
8727 8830 1
8728 8831 2 BEGIN
8729 8832 2
8730 8833 2 MAP
8731 8834 2     TOKENBUFFER: REF VECTOR[.BYTE]; ! Pointer to token Counted ASCII string
8732 8835 2
8733 8836 2 LOCAL
8734 8837 2     TOKEN: REF TOKEN$ENTRY;           ! Pointer to new Token Entry created
8735 8838 2
8736 8839 2
8737 8840 2
8738 8841 2     ! Get a temporary memory block for the Operator Token Entry.
8739 8842 2     ! Fill in the fields of the Token Entry and return its address.
8740 8843 2
8741 8844 2     TOKEN = DBG$GET_TEMPMEM(TOKENSK_ENTSIZE_OPERATOR + .TOKENBUFFER[0]/XUPVAL + 1);
8742 8845 2     TOKEN[TOKENSB_KIND] = .KIND;
8743 8846 2     TOKEN[TOKENSW_CODE] = .TOKEN_CODE;
8744 8847 2     CH$MOVE(.TOKENBUFFER[0] + 1, .TOKENBUFFER[0], TOKEN[TOKENSB_OPLEN]);
8745 8848 2     RETURN .TOKEN;
8746 8849 2
8747 8850 1 END;
```


				007C 00000 CREATE_OPERATOR_TOKEN:					
		50	08	BC	9A 00002	WORD	Save R2,R3,R4,R5,R6	:	8799
		50		04	C6 00006	MOVZBL	@TOKENBUFFER, R0	:	8844
			04	A0	9F 00009	DIVL2	#4, R0	:	
	00000000G	00		01	FB 0000C	PUSHAB	4(R0)	:	
		56		50	D0 00013	CALLS	#1, DBG\$GET_TEMPHEM	:	
		66	0C	AC	90 00016	MOVL	R0, TOKEN	:	
	02	A6	04	AC	B0 0001A	MOVB	KIND, (TOKEN)	:	8845
		50	08	BC	9A 0001F	MOVW	TOKEN CODE, 2(TOKEN)	:	8846
				50	D6 00023	MOVZBL	@TOKENBUFFER, R0	:	8847
0C	A6	08		50	28 00025	INCL	R0	:	
		50		56	D0 0002B	MOVC3	R0, @TOKENBUFFER, 12(TOKEN)	:	8848
				04	0002E	MOVL	TOKEN, R0	:	8850
						RET		:	

; Routine Size: 47 bytes, Routine Base: DBG\$CODE + 2504

```
8749 8851 1 ROUTINE CREATE_PRID_CONSTANT(PRID) =
8750 8852 1
8751 8853 1 FUNCTION
8752 8854 1 This routine creates a value descriptor for the predefined identifier
8753 8855 1 constant and returns the pointer to the value descriptor.
8754 8856 1
8755 8857 1 INPUTS
8756 8858 1 PRID - Pointer to a Predefined Identifier Constant entry in
8757 8859 1 Predefined Identifier Table for the given language.
8758 8860 1
8759 8861 1 OUTPUTS
8760 8862 1 A pointer to the value descriptor of Predefined Identifier Constant
8761 8863 1 is returned.
8762 8864 1
8763 8865 1
8764 8866 2 BEGIN
8765 8867 2
8766 8868 2 MAP
8767 8869 2 PRID: REF PRID$ENTRY; ! Pointer to Predefined Identifier
8768 8870 2
8769 8871 2 LOCAL
8770 8872 2 VALPTR: REF DBG$VALDESC; ! Pointer to Value Descriptor
8771 8873 2
8772 8874 2
8773 8875 2 VALPTR = DBG$MAKE_SKELETON_DESC(DBG$K_VALUE_DESC, 4);
8774 8876 2 VALPTR[DBG$B_DHDR_LANG] = DBG$GB_LANGUAGE;
8775 8877 2 VALPTR[DBG$B_DHDR_KIND] = RST$K_DATA;
8776 8878 2 VALPTR[DBG$B_DHDR_FCODE] = .PRID[PRID$B_FCODE];
8777 8879 2 VALPTR[DBG$B_VALUE_DTYPE] = .PRID[PRID$B_DTYPE];
8778 8880 2 VALPTR[DBG$B_VALUE_CLASS] = DBG$MAP_DTYPE_CLASS(
8779 8881 2 .PRID[PRID$B_DTYPE], FALSE);
8780 8882 2 VALPTR[DBG$W_VALUE_LENGTH] = DBG$NUM_BYTES(.PRID[PRID$B_DTYPE]);
8781 8883 2 VALPTR[DBG$S_VALUE_POINTER] = VALPTR[DBG$A_VALUE_ADDRESS];
8782 8884 2 VALPTR[DBG$S_VALUE_VALUE0] = .PRID[PRID$S_VALUE];
8783 8885 2 RETURN .VALPTR;
8784 8886 1 END;
```

```
000C 00000 CREATE_PRID_CONSTANT:
00000000G 7E 7A 04 DD 00002 .WORD Save R2,R3
03 A2 00000000G 8F 9A 00004 PUSHL #4
07 A2 02 FB 00008 MOVZBL #122, -(SP)
06 A2 52 50 D0 0000F CALLS #2, DBG$MAKE_SKELETON_DESC
16 A2 00 90 00012 MOVL R0, VALPTR
03 A2 06 90 0001A MOVB DBG$GB_LANGUAGE, 3(VALPTR)
07 A2 04 AC D0 0001E MOVB #6, 7(VALPTR)
06 A2 02 A3 90 00022 MOVL PRID, R3
16 A2 01 A3 90 00027 MOVB 2(R3), 6(VALPTR)
03 A2 7E D4 0002C MOVB 1(R3), 22(VALPTR)
07 A2 01 A3 9A 0002E CLRL -(SP)
00000000G 7E 01 A3 9A 0002E MOVZBL 1(R3), -(SP)
17 A2 02 FB 00032 CALLS #2, DBG$MAP_DTYPE_CLASS
07 A2 50 90 00039 MOVB R0, 23(VALPTR)
01 A3 9A 0003D MOVZBL 1(R3), -(SP)
```

```
8851
8875
8876
8877
8878
8879
8880
8881
8882
```

DBGPARSER
V04-000

1 3
16-Sep-1984 02:10:13
14-Sep-1984 12:17:30

VAX-11 BLISS-32 V4.0-742
[DEBUG.SRC]DBGPARSER.B32;1

Page 282
(32)

00000000G	00	01	FB	00041	CALLS	#1, DBG\$NUM_BYTES
14	A2	50	B0	00048	MOVW	R0, 20(VALPTR)
18	A2	20	A2	9E 0004C	MOVAB	32(VALPTR), 24(VALPTR)
20	A2	04	A3	D0 00051	MOVL	4(R3), 32(VALPTR)
	50		52	D0 00056	MOVL	VALPTR, R0
			04	00059	RET	

:
:
: 8883
: 8884
: 8885
: 8886

; Routine Size: 90 bytes, Routine Base: DBG\$CODE + 2533

```
8786 8887 1 ROUTINE DUMP_OPERATOR(OPERATOR, PRIMARY_FLAG): NOVALUE =
8787 8888 1
8788 8889 1 FUNCTION
8789 8890 1 This routine dumps out an Operator Lexical Token Entry in a readable
8790 8891 1 format when that operator is about to be evaluated. It prints one
8791 8892 1 line of output of approximately this format:
8792 8893 1
8793 8894 1 primary operator '\' evaluated (infix)
8794 8895 1
8795 8896 1 This routine is used only for internal DEBUG debugging purposes. Its
8796 8897 1 output is not seen by normal DEBUG users. It is only called when
8797 8898 1 Developer Switch 3 is set to trace operator evaluations as they occur.
8798 8899 1
8799 8900 1 INPUTS
8800 8901 1 OPERATOR - A pointer to the Lexical Token Entry for the operator
8801 8902 1 to be dumped.
8802 8903 1
8803 8904 1 PRIMARY_FLAG - A flag whose value is TRUE if this operator is
8804 8905 1 evaluated as part of a Primary Symbol. Its value is
8805 8906 1 FALSE if it is an ordinary expression operator.
8806 8907 1
8807 8908 1 OUTPUTS
8808 8909 1 NONE
8809 8910 1
8810 8911 1
8811 8912 2 BEGIN
8812 8913 2
8813 8914 2 MAP
8814 8915 2 OPERATOR: REF TOKEN$ENTRY; ! Pointer to operator's token entry
8815 8916 2
8816 8917 2
8817 8918 2
8818 8919 2 ! Print the main text of the message including the operator name string.
8819 8920 2 !
8820 8921 2 DBG$PRINT(UPLIT BYTE(%ASCIC ' '), 0);
8821 8922 2 IF .PRIMARY_FLAG THEN DBG$PRINT(UPLIT BYTE(%ASCIC 'primary '), 0);
8822 8923 2 DBG$PRINT(UPLIT BYTE(%ASCIC 'operator "'AC" evaluated ('),
8823 8924 2 OPERATOR[TOKEN$B_OPLEN]);
8824 8925 2
8825 8926 2
8826 8927 2 ! Then print what kind of operator this is.
8827 8928 2 !
8828 8929 2 IF .OPERATOR[TOKEN$B_KIND] EQL TOKEN$K_PREFIX_OP
8829 8930 2 THEN
8830 8931 2 DBG$PRINT(UPLIT BYTE(%ASCIC 'prefix'), 0)
8831 8932 2
8832 8933 2 ELSE IF .OPERATOR[TOKEN$B_KIND] EQL TOKEN$K infix_OP
8833 8934 2 THEN
8834 8935 2 DBG$PRINT(UPLIT BYTE(%ASCIC 'infix'), 0)
8835 8936 2
8836 8937 2 ELSE IF .OPERATOR[TOKEN$B_KIND] EQL TOKEN$K postfix_OP
8837 8938 2 THEN
8838 8939 2 DBG$PRINT(UPLIT BYTE(%ASCIC 'postfix'), 0)
8839 8940 2
8840 8941 2 ELSE
8841 8942 2 DBG$PRINT(UPLIT BYTE(%ASCIC 'invalid kind'), 0);
8842 8943 2
```



```
: 8843      8944 2
: 8844      8945 2
: 8845      8946 2
: 8846      8947 2
: 8847      8948 2
: 8848      8949 2
: 8849      8950 2
: 8850      8951 1

! Close out the message, flush the buffer, and return.
DBG$PRINT(UPLIT BYTE(XASCII ')'), 0);
DBG$NEWLINE();
RETURN;

END;
```

```
.PSECT DBG$PLIT,NOWRT, SHR, PIC,0

22 43 41 21 22 20 20 79 72 61 6D 69 20 20 04 03219 P.AXN: .ASCII <4>\ \
28 20 64 65 74 61 72 65 70 6F 1A 0321E P.AXO: .ASCII <8>\primary \
64 6E 69 68 20 78 69 66 65 72 70 06 03227 P.AXP: .ASCII <26>\operator "'!AC" evaluated (\
78 69 66 6E 69 05 03236 P.AXQ: .ASCII <6>\prefix\
78 69 66 6E 69 05 03242 P.AXQ: .ASCII <6>\prefix\
64 6E 69 68 20 78 69 66 6E 69 05 03249 P.AXR: .ASCII <5>\infix\
78 69 66 6E 69 05 0324F P.AXS: .ASCII <7>\postfix\
64 6E 69 68 20 78 69 66 6E 69 0C 03257 P.AXT: .ASCII <12>\invalid kind\
29 01 03264 P.AXU: .ASCII <1>\)\
```

```
.PSECT DBG$CODE,NOWRT, SHR, PIC,0

000C 00000 DUMP_OPERATOR:
53 00000000G 00 9E 00002 .WORD Save R2,R3
52 00000000' EF 9E 00009 MOVAB DBG$PRINT, R3
7E D4 00010 MOVAB P.AXN, R2
52 DD 00012 CLRL -(SP)
63 02 FB 00014 PUSHL R2
08 08 AC E9 00017 CALLS #2, DBG$PRINT
7E D4 00018 BLBC PRIMARY_FLAG, 1$
05 A2 9F 0001D CLRL -(SP)
63 02 FB 00020 PUSHAB P.AXO
7E 04 AC 0C C1 00023 1$: CALLS #2, DBG$PRINT
63 0E A2 9F 00028 ADDL3 #12, OPERATOR, -(SP)
02 04 BC 91 0002B PUSHAB P.AXP
07 12 00032 CALLS #2, DBG$PRINT
7E D4 00034 CMPB @OPERATOR, #2
29 A2 9F 00036 BNEQ 2$
1F 11 00039 CLRL -(SP)
03 04 BC 91 0003B 2$: PUSHAB P.AXQ
07 12 0003F BRB 5$
7E D4 00041 CMPB @OPERATOR, #3
30 A2 9F 00043 BNEQ 3$
12 11 00046 CLRL -(SP)
04 04 BC 91 00048 3$: PUSHAB P.AXR
07 12 0004C BRB 5$
7E D4 0004E CMPB @OPERATOR, #4
36 A2 9F 00050 BNEQ 4$
05 11 00053 CLRL -(SP)
7E D4 00055 4$: PUSHAB P.AXS
BRB 5$
CLRL -(SP)
```

```
8887
8921
8922
8924
8923
8924
8929
8931
8933
8935
8937
8939
8942
```

DBGPARSER
V04-000

L 3
16-Sep-1984 02:10:13
14-Sep-1984 12:17:30

VAX-11 Bliss-32 V4.0-742
[DEBUG.SRC]DBGPARSER.B32;1

Page 285
(33)

	63	3E	A2	9F	00057		PUSHAB	P.AXT
			02	FB	0005A	5%:	CALLS	#2, DBG\$PRINT
			7E	D4	0005D		CLRL	-(SP)
		4B	A2	9F	0005F		PUSHAB	P.AXU
	63		02	FB	00062		CALLS	#2, DBG\$PRINT
00000000G	00		00	FB	00065		CALLS	#0, DBG\$NEWLINE
			04	00	006C		RET	

..
8947
..
8948
..
8951

; Routine Size: 109 bytes, Routine Base: DBG\$CODE + 2580

```
8852 8952 1 ROUTINE DUMP_TOKEN(TOKEN): NOVALUE =
8853 8953 1
8854 8954 1 FUNCTION
8855 8955 1     This routine dumps out a specified Lexical Token Entry in a readable
8856 8956 1     format.  It prints one line of output of approximately this format:
8857 8957 1
8858 8958 1         token found: infix operator, code = 8, string = "**"
8859 8959 1
8860 8960 1     This routine is used only for internal DEBUG debugging purposes.
8861 8961 1     Its output is not seen by normal DEBUG users.  The routine is only
8862 8962 1     called by DBG$PRIMARY_PARSER if Developer Switch 2 is set.
8863 8963 1
8864 8964 1 INPUTS
8865 8965 1     TOKEN    - The address of the Lexical Token Entry to dump.
8866 8966 1
8867 8967 1 OUTPUTS
8868 8968 1     NONE
8869 8969 1
8870 8970 1 BEGIN
8871 8971 1
8872 8972 1 MAP
8873 8973 1     TOKEN: REF TOKEN$ENTRY;           ! Address of Lexical Token Entry to dump
8874 8974 1
8875 8975 1 LOCAL
8876 8976 1     KIND,                               ! Pointer to text saying what kind of
8877 8977 1     NAMEPTR,                           ! lexical token this is
8878 8978 1     PRIMARY;                           ! Pointer to the ASCII name string for
8879 8979 1                                         ! the lexical token
8880 8980 1                                         ! Pointer to string saying whether this
8881 8981 1                                         ! is a Primary Symbol operator
8882 8982 1
8883 8983 1 ! Determine what kind of Lexical Token Entry this is.
8884 8984 1 !
8885 8985 1 !
8886 8986 1 !
8887 8987 1 !
8888 8988 1 !
8889 8989 1 !
8890 8990 1 !
8891 8991 1 !
8892 8992 1 !
8893 8993 1 !
8894 8994 1 !
8895 8995 1 !
8896 8996 1 !
8897 8997 1 !
8898 8998 1 !
8899 8999 1 !
8900 9000 1 !
8901 9001 1 !
8902 9002 1 !
8903 9003 1 !
8904 9004 1 !
8905 9005 1 !
8906 9006 1 !
8907 9007 1 !
8908 9008 1 !
```

```

: 8909          9009          NAMEPTR = TOKEN[TOKEN$B_OPLEN];
: 8910          9010          END;
: 8911          9011
: 8912          9012
: 8913          9013          [TOKEN$K_POSTFIX_OP]:
: 8914          9014          BEGIN
: 8915          9015          KIND = UPLIT BYTE(%ASCII 'postfix operator');
: 8916          9016          NAMEPTR = TOKEN[TOKEN$B_OPLEN];
: 8917          9017          END;
: 8918          9018
: 8919          9019
: 8920          9020          [INRANGE, OUTFRANGE]:
: 8921          9021          BEGIN
: 8922          9022          KIND = UPLIT BYTE(%ASCII 'invalid kind');
: 8923          9023          NAMEPTR = UPLIT BYTE(%ASCII '');
: 8924          9024          END;
: 8925          9025
: 8926          9026          TES;
: 8927          9027
: 8928          9028
: 8929          9029          ! Print the line describing the Lexical Token Entry.
: 8930          9030          !
: 8931          9031          PRIMARY = UPLIT BYTE(%ASCII '');
: 8932          9032          IF .TOKEN[TOKEN$V_PRIMARY] THEN PRIMARY = UPLIT BYTE(%ASCII ' (primary)');
: 8933          9033          DBG$PRINT(UPLIT BYTE(%ASCII 'token found: !AC!AC, code = !SL, string = '!AC'''),
: 8934          9034          .KIND, .PRIMARY, .TOKEN[TOKEN$W_CODE], .NAMEPTR);
: 8935          9035          DBG$NEWLINE();
: 8936          9036          RETURN;
: 8937          9037
: 8938          9038          END;

```

```

                                .PSECT  DBG$PLIT,NOWRT,  SHR,  PIC,0
6F  74  61  72  65  70  6F  20  78  69  66  65  72  70  0F  03266 P.AXV:  .ASCII  <7>\operand\
72  6F  74  61  72  65  70  6F  20  78  69  66  6E  69  0E  0326E P.AXW:  .ASCII  <15>\prefix operator\
74  61  72  65  70  6F  20  78  69  66  74  73  6F  70  10  0327D
                                72  0327E P.AXX:  .ASCII  <14>\infix operator\
                                74  61  72  65  70  6F  20  78  69  66  74  73  6F  70  10  0328D P.AXY:  .ASCII  <16>\postfix operator\
                                72  6F  0329C
                                64  6E  69  6B  20  64  69  6C  61  76  6E  69  0C  0329E P.AXZ:  .ASCII  <12>\invalid kind\
                                00  032AB P.AYA:  .ASCII  <0>
                                00  032AC P.AYB:  .ASCII  <0>
21  20  3A  64  29  79  72  61  6D  69  72  70  28  20  0A  032AD P.AYC:  .ASCII  <10>\ (primary)\
21  20  3D  20  65  64  6F  66  20  6E  65  6B  6F  74  2F  032B8 P.AYD:  .ASCII  \token found: !AC!AC, code = !SL, string\
                                67  6E  69  72  74  73  20  2C  4C  53  032C7
                                22  43  41  21  22  20  3D  20  032D6
                                .ASCII  \ = '!AC'\

```

```

                                .PSECT  DBG$CODE,NOWRT,  SHR,  PIC,0
                                001C 00000 DUMP_TOKEN:
                                .WORD  Save R2,R3,R4
54 00000000' EF 9E 00002      MOVAB  P.AXZ, R4

```


0027	03 0021	51 01 001B	04 AC 61 0011	DO 8F 00009 0000D 00011 1\$:	MOVL CASEB .WORD	TOKEN, R1 (R1), #1, #3 2\$-1\$,- 3\$-1\$,- 4\$-1\$,- 5\$-1\$,-	8988
		53 50	64 9E 00019 0D A4 9E 0001C 1E 11 00020		MOVAB MOVAB BRB	P.AXZ, KIND P.AYA, NAMEPTR 7\$	9022 9023 8988
		53 50	C8 08 A4 9E 00022 2\$: A1 9E 00026 14 11 0002A		MOVAB MOVAB BRB	P.AXV, KIND 8(R1), NAMEPTR 7\$	8994 8995 8988
		53	DO A4 9E 0002C 3\$: 0A 11 00030		MOVAB BRB	P.AXW, KIND 6\$	9001 9002
		53	E0 A4 9E 00032 4\$: 04 11 00036		MOVAB BRB	P.AXX, KIND 6\$	9008 9009
		53 50 52 04 52	EF 0C 0E A1 9E 00038 5\$: A1 9E 0003C 6\$: A4 9E 00040 7\$: 01 A1 E9 00044		MOVAB MOVAB MOVAB BLBC	P.AXY, KIND 12(R1), NAMEPTR P.AYB, PRIMARY 1(R1), 8\$	9015 9016 9031 9032
		7E	0F A4 9E 00048 8\$: 50 DD 0004C 02 A1 3C 0004E 52 DD 00052 53 DD 00054		MOVAB PUSHL MOVZWL PUSHL PUSHL	P.AYC, PRIMARY NAMEPTR 2(R1), -(SP) PRIMARY KIND	9034
			1A A4 9F 00056 05 FB 00059 00 FB 00060 04 00067		PUSHAB CALLS CALLS RET	P.AYD #5, DBG\$PRINT #0, DBG\$NEWLINE	9033 9035 9038

; Routine Size: 104 bytes, Routine Base: DBG\$CODE + 251A

```
8940 9039 1 ROUTINE DUMP_PRIMARY(PRIMPTR): NOVALUE =
8941 9040 1
8942 9041 1 FUNCTION
8943 9042 1 This routine dumps out a Primary Descriptor or a Value Descriptor
8944 9043 1 in hexadecimal on the terminal. For a Primary Descriptor, it dumps
8945 9044 1 out not only the Root Node but also all Sub-Nodes. This routine is
8946 9045 1 used only for internal DEBUG debugging purposes. Its output is not
8947 9046 1 seen by normal DEBUG users. It is only called if Developer Switch
8948 9047 1 3 is set.
8949 9048 1
8950 9049 1 INPUTS
8951 9050 1 PRIMPTR - A pointer to the Primary Descriptor or the Value Descriptor
8952 9051 1 to be dumped out.
8953 9052 1
8954 9053 1 OUTPUTS
8955 9054 1 NONE
8956 9055 1
8957 9056 1 BEGIN
8958 9057 1
8959 9058 1 MAP
8960 9059 1 PRIMPTR: REF DBG$PRIMARY; ! Pointer to Primary Descriptor
8961 9060 1
8962 9061 1 LOCAL
8963 9062 1 LENGTH, ! Byte length of Primary Descr Sub-Node
8964 9063 1 NODEPTR: REF DBG$PRIM_NODE; ! Pointer to Primary Descr Sub-Node
8965 9064 1
8966 9065 1
8967 9066 1
8968 9067 1 ! If this is a Primary Descriptor, dump out the Root Node and then loop to
8969 9068 1 dump out all the individual Sub-Nodes.
8970 9069 1
8971 9070 1 IF .PRIMPTR[DBG$B_DHDR_TYPE] EQL DBG$K_PRIMARY_DESC
8972 9071 1 THEN
8973 9072 1 BEGIN
8974 9073 1
8975 9074 1 ! Dump out the Primary Descriptor Root Node.
8976 9075 1
8977 9076 1 !
8978 9077 1 DBG$DUMP_HEX(.PRIMPTR, DBG$K_PRIMARY_SIZE*%UPVAL, .PRIMPTR,
8979 9078 1 UPLIT BYTE (%ASCII ' Primary Descriptor:'));
8980 9079 1
8981 9080 1
8982 9081 1 ! Dump out each of the Primary Descriptor Sub-Nodes. Note that the
8983 9082 1 length of the Sub-Node in bytes depends on the FCODE stored in the
8984 9083 1 Sub-Node.
8985 9084 1
8986 9085 1 !
8987 9086 1 NODEPTR = .PRIMPTR[DBG$L PRIM FLINK];
8988 9087 1 WHILE .NODEPTR NEQ PRIMPTR[DBG$A_PRIM_FLINK] DO
8989 9088 1 BEGIN
8990 9089 1 IF .NODEPTR[DBG$B_PNODE_FCODE] EQL RST$K_TYPE_ARRAY
8991 9090 1 THEN
8992 9091 1 LENGTH = %UPVAL*(DBG$K PRIM SIZE ARRAY +
8993 9092 1 DBG$K_PRIM_SIZE_SUB5*.NODEPTR[DBG$B_PNARR_DIMCNT])
8994 9093 1
8995 9094 1 ELSE IF .NODEPTR[DBG$B_PNODE_FCODE] EQL RST$K_TYPE_RECORD
8996 9095 1 THEN
```

```

8997      9096 4      LENGTH = DBG$K_PRIM_SIZE_RECORD*%UPVAL
8998      9097 4
8999      9098 4      ELSE IF .NODEPTR[DBG$B_PNODE_FCODE] EQL RST$K_TYPE_VARIANT
9000      9099 4      THEN
9001      9100 4          LENGTH = DBG$K_PRIM_SIZE_VARIANT*%UPVAL
9002      9101 4
9003      9102 4      ELSE
9004      9103 4          LENGTH = DBG$K_PRIM_SIZE_NORMAL*%UPVAL;
9005      9104 4
9006      9105 4      DBG$DUMP_HEX(.NODEPTR, .LENGTH, .NODEPTR,
9007      9106 4          UPLIT BYTE (%ASCII ' Primary Sub-Node:'));
9008      9107 4      NODEPTR = .NODEPTR[DBG$L_PNODE_FLINK];
9009      9108 4      END;
9010      9109 4
9011      9110 4      END
9012      9111 4
9013      9112 4      ! If this is a Value Descriptor, dump it as such.
9014      9113 4      !
9015      9114 4      ELSE IF .PRIMPTR[DBG$B_DHDR_TYPE] EQL DBG$K_VALUE_DESC
9016      9115 4      THEN
9017      9116 4          DBG$DUMP_HEX(.PRIMPTR, .PRIMPTR[DBG$W_DHDR_LENGTH], .PRIMPTR,
9018      9117 4              UPLIT BYTE (%ASCII ' Value Descriptor:'))
9019      9118 4
9020      9119 4      ! If this is a Volatile Value Descriptor, dump it as such.
9021      9120 4      !
9022      9121 4      ELSE IF .PRIMPTR[DBG$B_DHDR_TYPE] EQL DBG$K_V_VALUE_DESC
9023      9122 4      THEN
9024      9123 4          DBG$DUMP_HEX(.PRIMPTR, .PRIMPTR[DBG$W_DHDR_LENGTH], .PRIMPTR,
9025      9124 4              UPLIT BYTE (%ASCII ' Volatile Value Descriptor:'))
9026      9125 4
9027      9126 4      ! If it is none of the above, something is wrong but we try to dump the
9028      9127 4      ! descriptor anyway.
9029      9128 4      !
9030      9129 4      ELSE
9031      9130 4          DBG$DUMP_HEX(.PRIMPTR, .PRIMPTR[DBG$W_DHDR_LENGTH], .PRIMPTR,
9032      9131 4              UPLIT BYTE (%ASCII ' Invalid Descriptor Type:'));
9033      9132 4
9034      9133 4      ! We are all done--now return.
9035      9134 4      !
9036      9135 4      !
9037      9136 4      !
9038      9137 4      !
9039      9138 4      !
9040      9139 4      RETURN;
9041      9140 4      END;
```

```

.PSECT DBG$PLIT, NOWRT, SHR, PIC, 0
20 79 72 61 6D 69 72 50 20 20 20 20 20 20 19 032E8 P.AYE: .ASCII <25>\ Primary Descriptor:\
20 79 72 61 6D 69 72 50 20 20 20 20 20 20 17 032F7 P.AYF: .ASCII <23>\ Primary Sub-Node:\
65 44 20 65 75 6C 61 56 20 20 20 20 20 20 17 03311 P.AYG: .ASCII <23>\ Value Descriptor:\
65 6C 69 74 61 6C 6F 56 20 20 20 20 20 20 20 03329 P.AYH: .ASCII \ Volatile Value Descriptor:\
20 79 72 61 6D 69 72 50 20 20 20 20 20 20 17 03332
```

74	70	69	72	63	73	65	44	20	65	75	6C	61	56	20	03341
20	64	69	6C	61	76	6E	49	20	20	20	20	3A	72	6F	03350
65	70	79	54	20	72	6F	74	70	69	72	63	73	65	44	03353
														3A	03362
															03371

P.AYI: .ASCII <30>\

Invalid Descriptor Type:\

.PSECT DBG\$CODE,NOWRT, SHR, PIC.0

007C 00000 DUMP_PRIMARY:							
	56	00000000G	00	9E	00002	.WORD Save R2,R3,R4,R5,R6	9039
	55	000000000	EF	9E	00009	MOVAB DBGSDUMP HEX, R6	
	54	04	AC	D0	00010	MOVAB P.AYE, R5	
79	8F	02	A4	91	00014	MOVL PRIMPTR, R4	9071
			51	12	00019	CMPI 2(R4), #121	
			30	BB	0001B	BNEQ 6\$	
			24	DD	0001D	PUSHR #M<R4,R5>	9078
			54	DD	0001F	PUSHL #36	
	66		04	FB	00021	PUSHL R4	
	52	14	A4	D0	00024	CALLS #4, DBGSDUMP HEX	
	50	14	A4	9E	00028	MOVL 20(R4), NODEPTR	9086
	50		52	D1	0002C	MOVAB 20(R4), R0	9087
			60	13	0002F	CMPL NODEPTR, R0	
	01	09	A2	91	00031	BEQL 10\$	
			0D	12	00035	CMPI 9(NODEPTR), #1	9089
	50	1B	A2	9A	00037	BNEQ 2\$	
	50		14	C4	0003B	MOVZBL 27(NODEPTR), R0	9092
	53	28	A0	9E	0003E	MULL2 #20, R0	
			19	11	00042	MOVAB 40(R0), LENGTH	9091
	07	09	A2	91	00044	BRB 5\$	
			05	12	00048	CMPI 9(NODEPTR), #7	9094
	53		1C	D0	0004A	BNEQ 3\$	
			0E	11	0004D	MOVL #28, LENGTH	9096
	13	09	A2	91	0004F	BRB 5\$	
			05	12	00053	CMPI 9(NODEPTR), #19	9098
	53		28	D0	00055	BNEQ 4\$	
			03	11	00058	MOVL #40, LENGTH	9100
	53		18	D0	0005A	BRB 5\$	
		1A	A5	9F	0005D	MOVL #24, LENGTH	9103
			52	DD	00060	PUSHAB P.AYF	9106
			0C	BB	00062	PUSHL NODEPTR	9105
	66		04	FB	00064	PUSHR #M<R2,R3>	
	52		62	D0	00067	CALLS #4, DBGSDUMP HEX	
			BC	11	0006A	MOVL (NODEPTR), NODEPTR	9107
7A	8F	02	A4	91	0006C	BRB 1\$	9087
			05	12	00071	CMPI 2(R4), #122	9115
		32	A5	9F	00073	BNEQ 7\$	
			0F	11	00076	PUSHAB P.AYG	9118
			0F	11	00076	BRB 9\$	9117
83	8F	02	A4	91	00078	CMPI 2(R4), #131	9123
			05	12	0007D	BNEQ 8\$	
		4A	A5	9F	0007F	PUSHAB P.AYH	9126
			03	11	00082	BRB 9\$	9125
		6B	A5	9F	00084	PUSHAB P.AYI	9134
			54	DD	00087	PUSHL R4	9133
	7E		64	3C	00089	MOVZWL (R4), -(SP)	

DBGPARSER
V04-000

F 4
16-Sep-1984 02:10:13
14-Sep-1984 12:17:30

VAX-11 Bliss-32 V4.0-742
[DEBUG.SRC]DBGPARSER.B32;1

Page 292
(35)

66

54 DD 0008C PUSHL R4
04 FB 0008E CALLS #4, DBGSDUMP_HEX
04 00091 108: RET

:
:
: 9140

; Routine Size: 146 bytes, Routine Base: DBG\$CODE + 2662

```
9043 9141 1 ROUTINE FIX_UP_PRIMARY(PRIMPTR): NOVALUE =
9044 9142 1
9045 9143 1 FUNCTION
9046 9144 1 This routine is needed to handle a special case that may arise
9047 9145 1 in conjunction with the array slice feature: If X is a two -
9048 9146 1 dimensional array, say 10x10, and the user specifies EX X[5],
9049 9147 1 then we want to treat this as if he had said X[5][1:10].
9050 9148 1 However, we do not know at the time we are picking up the "5"
9051 9149 1 subscript whether he is going to say X[5] or X[5][6], for example.
9052 9150 1 So we cannot fix up the lower/upper bounds properly until after
9053 9151 1 all the subscripts have been picked up. This is the routine
9054 9152 1 that gets called after all the subscripts have been picked
9055 9153 1 up, to fix up these bounds.
9056 9154 1
9057 9155 1 INPUTS
9058 9156 1 PRIMPTR - A pointer to the Primary Descriptor being constructed.
9059 9157 1
9060 9158 1 OUTPUTS
9061 9159 1 The Primary Descriptor pointed to by PRIMPTR may be modified.
9062 9160 1
9063 9161 2 BEGIN
9064 9162 2 MAP
9065 9163 2 PRIMPTR: REF DBG$PRIMARY;
9066 9164 2
9067 9165 2 LOCAL
9068 9166 2 NODEPTR: REF DBG$PRIM_NODE,
9069 9167 2 SUBVECTOR: REF DBG$PRIM_NODE_SUBS;
9070 9168 2
9071 9169 2
9072 9170 2 ! Obtain a pointers to the Primary Descriptor Subnode and the
9073 9171 2 ! subscript vector within that subnode. Do nothing if the
9074 9172 2 ! subnode is not for an array.
9075 9173 2
9076 9174 2 NODEPTR = .PRIMPTR[DBG$L PRIM_BLINK];
9077 9175 2 IF .NODEPTR[DBG$B_PNODE_FCODE] NEQ RST$K_TYPE_ARRAY
9078 9176 2 THEN
9079 9177 2 RETURN;
9080 9178 2 SUBVECTOR = NODEPTR[DBG$A_PNARR_SVECTOR];
9081 9179 2
9082 9180 2
9083 9181 2 ! Check for the subscript count being less than the dimension count,
9084 9182 2 ! but the range bit not being set. The point here is that we want
9085 9183 2 ! to treat this case as if it really were a range (e.g., treat
9086 9184 2 ! X[1] as being the same as X[1][1:10]. Note that this same code
9087 9185 2 ! appears in GET_SUBSCRIPTS when we discover that we do have a range.
9088 9186 2
9089 9187 2 IF .NODEPTR[DBG$B_PNARR_SUBCNT] LSS .NODEPTR[DBG$B_PNARR_DIMCNT]
9090 9188 2 AND NOT .NODEPTR[DBG$V_PNARR_RANGE]
9091 9189 2 THEN
9092 9190 2 BEGIN
9093 9191 2 NODEPTR[DBG$V_PNARR_RANGE] = TRUE;
9094 9192 2 INCR I FROM 0 TO .NODEPTR[DBG$B_PNARR_SUBCNT] - 1 DO
9095 9193 2 BEGIN
9096 9194 2 SUBVECTOR[I, DBG$L PNSUB_LBOUND] =
9097 9195 2 SUBVECTOR[I, DBG$L PNSUB_SVALUE];
9098 9196 2 SUBVECTOR[I, DBG$L PNSUB_UBOUND] =
9099 9197 2 SUBVECTOR[I, DBG$L PNSUB_SVALUE];
```

; Routine Size: 74 bytes, Routine Base: DBG\$CODE + 26F4

```
9105 9202 1 ROUTINE GET_BLISS_SUBSCRIPTS(PRIMPTR, NAME): NOVALUE =
9106 9203 1
9107 9204 1 FUNCTION
9108 9205 1 This routine picks up subscript values in a BLISS structure
9109 9206 1 reference (i.e., any BLISS primary of the form X[...]).
9110 9207 1 It calls DBG$EXPRESSION_PARSER to pick up each subscript.
9111 9208 1
9112 9209 1 The kinds of BLISS structures we accept are:
9113 9210 1
9114 9211 1 X[i] bitvector
9115 9212 1 X[i] vector
9116 9213 1 X[n,p,s,e] block
9117 9214 1 X[m,n,p,s,e] blockvector
9118 9215 1
9119 9216 1 This routine assumes that the opening subscript bracket has
9120 9217 1 already been found and that the parse pointer points to the start
9121 9218 1 of the first subscript expression. When this routine returns,
9122 9219 1 the parse pointer is left pointing at the first character after
9123 9220 1 the closing subscript bracket.
9124 9221 1
9125 9222 1 INPUTS
9126 9223 1 PRIMPTR - A pointer to the Primary Descriptor for a structure about
9127 9224 1 to be subscripted.
9128 9225 1 NAME - The name of the object being subscripted (for error
9129 9226 1 message purposes)
9130 9227 1
9131 9228 1 OUTPUTS
9132 9229 1 The PRIMPTR Primary Descriptor is changed to include the subscript
9133 9230 1 information. This is represented as follows:
9134 9231 1
9135 9232 1 X[i] bitvector - i is stored in a Primary Descriptor Array
9136 9233 1 sub-node.
9137 9234 1 X[i] vector - i is stored in a Primary Descriptor Array
9138 9235 1 sub-node.
9139 9236 1 X[n,p,s,e] block - n is stored in a Primary Descriptor Array
9140 9237 1 sub-node. p, s, and e are stored in the
9141 9238 1 Primary Descriptor Root node, in the
9142 9239 1 prim_offset, prim_length, and sgnext fields.
9143 9240 1 X[m,n,p,s,e] blockvector- m and n are stored in a Primary Descriptor
9144 9241 1 Array sub-node. p, s, and e are stored in the
9145 9242 1 Primary Descriptor Root node, in the
9146 9243 1 prim_offset, prim_length, and sgnext fields.
9147 9244 1
9148 9245 1 BEGIN
9149 9246 1
9150 9247 1 MAP
9151 9248 1 PRIMPTR: REF DBG$PRIMARY; ! Pointer to BLISS structure Primary
9152 9249 1 ! Descriptor.
9153 9250 1
9154 9251 1 LOCAL
9155 9252 1 COUNT, ! Count of field values
9156 9253 1 DECLTYPE: REF DBG$VALDESC, ! Pointer to value descriptor for the
9157 9254 1 ! subscript value.
9158 9255 1 DSTPTR: REF DST$RECORD, ! Pointer to BLISS structure DST entry
9159 9256 1 FCODE, ! Data type FCODE for current symbol
9160 9257 1 LA_PTR: REF VECTOR[.BYTE], ! Lookahead pointer into input
9161 9258 2 LOW_RANGE_VAL, ! The lower value of a subscript range
```



```

9162 9259 2      NODEPTR: REF DBG$PRIM NODE, ! Pointer to Prim Desc Array sub-node
9163 9260      NODESUBPTR: REF DBG$PRIM_NODE_SUBS, ! Pointer to subscript blockvector
9164 9261                                     in Prim Desc Array sub-node
9165 9262      PTR: REF VECTOR[.LONG], ! Temporary pointer to field name values
9166 9263      REF FLAG, ! True for REF objects
9167 9264      RSTPTR: REF RST$ENTRY, ! Pointer to BLISS structure RST entry
9168 9265      SAVED RADIX, ! Temporarily saved expression radix
9169 9266      STRIDE, ! Stride for blocks and blockvectors
9170 9267      STRUC, ! Code for kind of BLISS structure
9171 9268      SUBSCR COUNT, ! Count of the number of subscripts
9172 9269      SUBVECTOR: VECTOR[5], ! Vector of subscripts
9173 9270      TOKEN, ! Pointer to a Lexical Token
9174 9271      TYPEID, ! Pointer to a typeid
9175 9272      VALPTR: REF DBG$VALDESC, ! Pointer to subscript Value Descriptor
9176 9273      VALUE: ! Subscript value
9177 9274
9178 9275      DBG$GL_CURRENT_PRIMARY = .PRIMPTR;
9179 9276
9180 9277      ! Check that the Primary Descriptor has the correct Kind and FCODE.
9181 9278      ! If so, obtain a pointer to the DST record for this BLISS structure.
9182 9279
9183 9280      RSTPTR = .PRIMPTR [DBG$GL_DHDR_SYMID0];
9184 9281      IF .RSTPTR EQL 0
9185 9282      THEN
9186 9283          SIGNAL(DBG$NOTASTRUCT, 1, .NAME);
9187 9284      DSTPTR = .RSTPTR [RST$DSTPTR];
9188 9285      IF .PRIMPTR [DBG$B_DHDR_KIND] NEQ RST$K_DATA
9189 9286      THEN
9190 9287          SIGNAL(DBG$NOTASTRUCT, 1, DSTPTR[DST$B_NAME]);
9191 9288
9192 9289      DBG$STA SYMTYPE (.RSTPTR, FCODE, TYPEID);
9193 9290      IF .FCODE NEQ RST$K_TYPE_BLIDATA
9194 9291      THEN
9195 9292          SIGNAL(DBG$NOTASTRUCT, 1, DSTPTR[DST$B_NAME]);
9196 9293
9197 9294
9198 9295      ! Check for no field structure in the RST - this arises when the
9199 9296      ! user defines his own BLISS structure, instead of using one of
9200 9297      ! the builtins BITVECTOR, VECTOR, BLOCK, or BLOCKVECTOR. We do
9201 9298      ! not handle this case.
9202 9299
9203 9300      STRUC = .DSTPTR [DST$V_BLI_STRUC];
9204 9301      IF .STRUC EQL DST$K_BLI_NOSTRUC
9205 9302      THEN
9206 9303          SIGNAL(DBG$NOTASTRUCT, 1, DSTPTR[DST$B_NAME]);
9207 9304
9208 9305
9209 9306      ! If the REF bit is set, then there is a sub-node for the dereferencing.
9210 9307      ! Call DBG$BUILD_PRIMARY_SUBNODE to build a new subnode for the array
9211 9308      ! information. Light the EVAL bit in the dereference subnode.
9212 9309
9213 9310      IF .DSTPTR [DST$V_BLI_REF]
9214 9311      THEN
9215 9312          BEGIN
9216 9313              DBG$BUILD_PRIMARY_SUBNODE (.PRIMPTR, RST$K_DATA, 0, RST$K_TYPE_BLIDATA,
9217 9314                  .PRIMPTR [DBG$GL_DHDR_TYPEID], .DSTPTR);
9218 9315              NODEPTR = .PRIMPTR [DBG$GL_PRIM_FLINK];

```

```

9219
9220
9221
9222
9223
9224
9225
9226
9227
9228
9229
9230
9231
9232
9233
9234
9235
9236
9237
9238
9239
9240
9241
9242
9243
9244
9245
9246
9247
9248
9249
9250
9251
9252
9253
9254
9255
9256
9257
9258
9259
9260
9261
9262
9263
9264
9265
9266
9267
9268
9269
9270
9271
9272
9273
9274
9275

```

```

NODEPTR [DBG$V_PNODE_EVAL] = TRUE;
REF_FLAG = TRUE;
END
ELSE
REF_FLAG = FALSE;

! Obtain pointers to the Primary Descriptor Sub-Node and the subscript
! blockvector within that node. Light the eval bit in the subnode.
NODEPTR = .PRIMPTR [DBG$L PRIM BLINK];
NODESUBPTR = NODEPTR [DBG$A PNARR_SVECTOR];
NODEPTR [DBG$V_PNODE_EVAL] = TRUE;

! Loop through the list of subscripts. Pick up each actual subscript and
! add it to the subscript vector. When we get to the closing subscript
! parenthesis, we check all the subscript values for validity and complete
! the Primary Descriptor Sub-Node accordingly.
SUBSCR_COUNT = 0;
TERMINATOR_CODE = TOKEN$K_TERM_COMMA;
WHILE .TERMINATOR_CODE NEQ TOKEN$K_TERM_CLOSE DO
BEGIN

! Look for the asterisk. X[*] is the same as X[lower:upper].
! If we find the asterisk then advance the character pointer beyond
! the asterisk and also increment the subscript count.
LA_PTR = .CHARPTR;
WHILE .LA_PTR[0] EQL ' ' DO LA_PTR = .LA_PTR + 1;
IF .LA_PTR[0] EQL '*'
THEN
BEGIN
IF .NODEPTR[DBG$V_PNARR_RANGE] OR
(.SUBSCR_COUNT NEQ 0) OR
((.STRUC NEQ DST$K_BLI_BITVEC) AND
(.STRUC NEQ DST$K_BLI_VEC) AND
(.STRUC NEQ DST$K_BLI_BLKVEC))
THEN
SIGNAL(DBG$ _INVRANSPEC);

CHARPTR = .LA_PTR + 1;
SUBSCR_COUNT = .SUBSCR_COUNT + 1;

! Call the Lexical Scanner to take us past the ',', ' or
! or ']' or ')'. This will set TERMINATOR_CODE to the
! terminator that is seen. If we do not see a terminator
! then signal a syntax error. Also signal an error if
! ':' was the terminator.
TOKEN = DBG$LEXICAL_SCANNER (FALSE, FALSE,
.SUBSCRIPT_TERM_TBL, 0);
IF .TOKEN NEQ TERMINATOR_TOKEN
THEN
BEGIN

```

```
9276 9373 5 LOCAL
9277 9374 5 ASCIC_STRING: VECTOR[2,BYTE];
9278 9375 5 ASCIC_STRING[0] = 1;
9279 9376 5 ASCIC_STRING[1] = .CHARPTR[0];
9280 9377 5 SIGNAL(DBG$_SYNERREXPR, 1, ASCIC_STRING);
9281 9378 4 END;
9282 9379 4 IF .TERMINATOR_CODE EQL TOKEN$K_TERM_COLON
9283 9380 4 THEN
9284 9381 4 SIGNAL(DBG$_INVTRANSPEC);
9285 9382 4 IF .TERMINATOR_CODE EQL TOKEN$K_TERM_NONE
9286 9383 4 THEN
9287 9384 4 SIGNAL(DBG$_MISCLOSUB);
9288 9385 4 CHARPTR = .CHARPTR + .TERMINATOR_LENGTH;
9289 9386 4
9290 9387 4 ! Turn this reference into a range.
9291 9388 4 !
9292 9389 4 NODEPTR[DBG$V_PNARR_RANGE] = TRUE;
9293 9390 4 LOW_RANGE_VAL = .NODESUBPTR[0, DBG$L_PNSUB_LBOUND];
9294 9391 4 SUBVECTOR[0] = .NODESUBPTR[0, DBG$L_PNSUB_UBOUND];
9295 9392 4 END
9296 9393 4
9297 9394 4 ELSE
9298 9395 3 BEGIN
9299 9396 4
9300 9397 4 ! Call the Expression Parser to pick up the next subscript. Note that
9301 9398 4 ! we set the radix to decimal over this call and then restore it. Also
9302 9399 4 ! note that OWN variables TERMINATOR_CODE and TERMINATOR_LENGTH are set
9303 9400 4 ! by the Expression Parser as side-effects.
9304 9401 4
9305 9402 4 SAVED_RADIX = .EXPRESSION_RADIX;
9306 9403 4 EXPRESSION_RADIX = DBG$K_DECIMAL;
9307 9404 4 VALPTR = DBG$EXPRESSION_PARSER(FALSE, .SUBSCRIPT_TERM_TBL);
9308 9405 4 EXPRESSION_RADIX = .SAVED_RADIX;
9309 9406 4
9310 9407 4
9311 9408 4 ! Check the terminator code. If there was no terminator (i.e., the
9312 9409 4 ! input line just ended), signal an error. Otherwise we got a comma,
9313 9410 4 ! colon, or closing bracket and we increment CHARPTR to get past it.
9314 9411 4
9315 9412 4 IF .TERMINATOR_CODE EQL TOKEN$K_TERM_NONE THEN SIGNAL(DBG$_MISCLOSUB);
9316 9413 4 CHARPTR = .CHARPTR + .TERMINATOR_LENGTH;
9317 9414 4
9318 9415 4
9319 9416 4 ! Check whether this "subscript" is actually a BLISS field name. If
9320 9417 4 ! so, set up a pointer to the BLISS field components and copy those
9321 9418 4 ! four components into the subscript vector.
9322 9419 4
9323 9420 4 IF .VALPTR[DBG$B_DHDR_FCODE] EQL RST$K_TYPE_BLIFLD
9324 9421 4 THEN
9325 9422 4 BEGIN
9326 9423 4 IF .NODEPTR[DBG$V_PNARR_RANGE] AND (.SUBSCR_COUNT EQL 0)
9327 9424 4 THEN
9328 9425 4 SIGNAL(DBG$_INVTRANSPEC);
9329 9426 4
9330 9427 4 PTR = VALPTR[DBG$A_VALUE_ADDRESS];
9331 9428 4
9332 9429 4
```



```

9333 9430 5
9334 9431 5
9335 9432 6
9336 9433 6
9337 9434 6
9338 9435 6
9339 9436 6
9340 9437 6
9341 9438 6
9342 9439 6
9343 9440 6
9344 9441 6
9345 9442 6
9346 9443 6
9347 9444 6
9348 9445 6
9349 9446 6
9350 9447 6
9351 9448 6
9352 9449 6
9353 9450 6
9354 9451 6
9355 9452 6
9356 9453 6
9357 9454 6
9358 9455 6
9359 9456 6
9360 9457 6
9361 9458 6
9362 9459 6
9363 9460 6
9364 9461 6
9365 9462 6
9366 9463 6
9367 9464 6
9368 9465 6
9369 9466 6
9370 9467 6
9371 9468 6
9372 9469 6
9373 9470 6
9374 9471 6
9375 9472 6
9376 9473 6
9377 9474 6
9378 9475 6
9379 9476 6
9380 9477 6
9381 9478 6
9382 9479 6
9383 9480 6
9384 9481 6
9385 9482 6
9386 9483 6
9387 9484 6
9388 9485 6
9389 9486 6

```

```

COUNT = .PTR[0];
INCR 1 FROM 1 TO .COUNT DO
    BEGIN
        IF .SUBSCR_COUNT LSS 5 THEN SUBVECTOR[.SUBSCR_COUNT] = .PTR[.I];
        SUBSCR_COUNT = .SUBSCR_COUNT + 1;
    END;
END

! The subscript is not a BLISS field name so we pick up the subscript
! value and convert it to integer. This subscript value is then put
! away in the subscript vector and the subscript count is incremented.
ELSE
    BEGIN

        ! Convert the value descriptor to an integer value.
        VALUE = CONVERT_TO_INTEGER (.VALPTR);

        ! If the terminator is a colon, we have the first part of a sub-
        ! script range specification (such as 'ARR[2:5]'). See if a range
        ! specification is allowed (it is only allowed on the first sub-
        ! script of a bitvector, a normal vector, or a blockvector) and
        ! save away this lower value of the range.
        IF .TERMINATOR_CODE EQL TOKEN$K_TERM_COLON
        THEN
            BEGIN
                IF .NODEPTR[DBG$V_PNARR_RANGE] OR
                    (.SUBSCR_COUNT NEQ 0) OR
                    ((.STRUC NEQ DST$K_BLI_BITVEC) AND
                     (.STRUC NEQ DST$K_BLI_VEC) AND
                     (.STRUC NEQ DST$K_BLI_BLKVEC))
                THEN
                    SIGNAL(DBG$_INVRANSPEC);

                    NODEPTR[DBG$V_PNARR_RANGE] = TRUE;
                    LOW_RANGE_VAL = .VALUE;
                END
            END

            ! The terminator is not a colon, so we have the complete specifica-
            ! tion for the current subscript. Here we simply fill the subscript
            ! value (or the upper bound of the range) into SUBVECTOR and incre-
            ! ment the subscript count. We can ignore subscripts after the
            ! fifth one because they will be in error anyway.
        ELSE
            BEGIN
                IF .SUBSCR_COUNT LSS 5
                THEN
                    SUBVECTOR[.SUBSCR_COUNT] = .VALUE;
            END
        END
    END

```



```
9390 9487
9391 9488
9392 9489
9393 9490
9394 9491
9395 9492
9396 9493
9397 9494
9398 9495
9399 9496
9400 9497
9401 9498
9402 9499
9403 9500
9404 9501
9405 9502
9406 9503
9407 9504
9408 9505
9409 9506
9410 9507
9411 9508
9412 9509
9413 9510
9414 9511
9415 9512
9416 9513
9417 9514
9418 9515
9419 9516
9420 9517
9421 9518
9422 9519
9423 9520
9424 9521
9425 9522
9426 9523
9427 9524
9428 9525
9429 9526
9430 9527
9431 9528
9432 9529
9433 9530
9434 9531
9435 9532
9436 9533
9437 9534
9438 9535
9439 9536
9440 9537
9441 9538
9442 9539
9443 9540
9444 9541
9445 9542
9446 9543
```

```
        SUBSCR_COUNT = .SUBSCR_COUNT + 1;
      END;
    END;
  END;
    ! End of WHILE loop over subscripts

! We found the end of the subscript list, i.e. the closing subscript
! parenthesis. Now case on the kind of BLISS structure we are dealing
! with and check the subscript values and build the Primary Descriptor
! Sub-Node accordingly.
CASE .STRUC FROM DST$K_BLI_NOSTRUC TO DST$K_BLI_BLKVEC OF
  SET

    ! Handle the Bitvector case.
    [DST$K_BLI_BITVEC]:
      BEGIN

        ! Check that the bit-vector had exactly one subscript.
        IF .SUBSCR_COUNT LSS 1 THEN SIGNAL(DBG$_TOOFEWSUB, 1, 1);
        IF .SUBSCR_COUNT GTR 1 THEN SIGNAL(DBG$_TOOMANSUB, 1, 1);

        ! Check that the subscript value is in range.
        IF NOT .REF_FLAG
        THEN
          BEGIN
            IF (.SUBVECTOR[0] LSS 0) OR
                (.SUBVECTOR[0] GEQ .DSTPTR[DST$L_BLI_BITVEC_SIZE])
            THEN
              SIGNAL (DBG$ STRUCSIZE, 2,
                      .DSTPTR[DST$L_BLI_BITVEC_SIZE], .SUBVECTOR[0]);

              ! If a subscript range was specified, check that the lower range
              ! value is also in range.
              IF .NODEPTR[DBG$V_PNARR_RANGE] AND
                  ((.LOW_RANGE_VAL LSS 0) OR
                   (.LOW_RANGE_VAL GEQ .DSTPTR[DST$L_BLI_BITVEC_SIZE]))
              THEN
                SIGNAL (DBG$ STRUCSIZE, 2,
                        .DSTPTR[DST$L_BLI_BITVEC_SIZE], .LOW_RANGE_VAL);
              END;

              ! Fill in the Primary Descriptor Array Sub-Node.
              NODEPTR [DBG$B_PNARR_SUBCNT] = 1;
              NODESUBPTR [0, DBG$L_PNSUB_SVALUE] = .SUBVECTOR[0];
            END;
          END;
        END;
      END;
    END;
  END;
END;
```

```
9447 9544
9448 9545      END;                                ! End of bitvector case
9449 9546
9450 9547
9451 9548      ! Handle ordinary BLISS vectors.
9452 9549      [DST$K_BLI_VEC]:
9453 9550      BEGIN
9454 9551
9455 9552
9456 9553      ! Check that the vector had exactly one subscript.
9457 9554
9458 9555      IF .SUBSCR_COUNT LSS 1 THEN SIGNAL(DBG$_TOOFEWSUB, 1, 1);
9459 9556      IF .SUBSCR_COUNT GTR 1 THEN SIGNAL(DBG$_TOOMANSUB, 1, 1);
9460 9557
9461 9558
9462 9559      ! Check that the subscript value is in range.
9463 9560
9464 9561      IF NOT .REF_FLAG
9465 9562      THEN
9466 9563      BEGIN
9467 9564          IF (.SUBVECTOR[0] LSS 0) OR
9468 9565             (.SUBVECTOR[0] GEQ .DSTPTR[DST$L_BLI_VEC_UNITS])
9469 9566          THEN
9470 9567              SIGNAL(DBG$_STRUFSIZE, 2,
9471 9568                  .DSTPTR[DST$L_BLI_VEC_UNITS], .SUBVECTOR[0]);
9472 9569
9473 9570
9474 9571      ! If a subscript range was specified, check that the lower range
9475 9572      ! value is also in range.
9476 9573
9477 9574      IF .NODEPTR[DBG$V_PNARR_RANGE] AND
9478 9575         ((.LOW_RANGE_VAL LSS 0) OR
9479 9576          (.LOW_RANGE_VAL GEQ .DSTPTR[DST$L_BLI_VEC_UNITS]))
9480 9577      THEN
9481 9578          SIGNAL(DBG$_STRUFSIZE, 2,
9482 9579              .DSTPTR[DST$L_BLI_VEC_UNITS], .LOW_RANGE_VAL);
9483 9580
9484 9581      END;
9485 9582
9486 9583      ! Fill in the Primary Descriptor Sub-Node.
9487 9584
9488 9585      NODEPTR [DBG$B_PNARR_SUBCNT] = 1;
9489 9586      NODESUBPTR [0, DBG$L_PNSUB_SVALUE] = .SUBVECTOR[0];
9490 9587
9491 9588      END;                                ! End of BLISS vector case
9492 9589
9493 9590
9494 9591
9495 9592      ! Handle BLISS blocks.
9496 9593
9497 9594      [DST$K_BLI_BLOCK]:
9498 9595      BEGIN
9499 9596
9500 9597      ! Fill in the correct information for the array subnode
9501 9598      ! (It was dummied up as an array of longwords for purposes
9502 9599      ! of aggregate output earlier).
9503 9600
```

```
9504 9601 STRIDE = .DSTPTR [DST$V_BLI_BLOCK_UNIT_SIZE];
9505 9602 NODESUBPTR [0, DBG$PNSUB_STRIDE] = .STRIDE;
9506 9603 NODESUBPTR [0, DBG$PNSUB_UBOUND] =
9507 9604 .DSTPTR [DST$L_BLI_BLOCK_UNITS]-1;
9508 9605 NODEPTR[DBG$W_PNARR_LENGTH] = .STRIDE;
9509 9606 IF .STRIDE EQ 1
9510 9607 THEN
9511 9608 IF .SUBVECTOR[3]
9512 9609 THEN
9513 9610 NODEPTR [DBG$B_PNARR_DTYPE] = DSC$K_DTYPE_B
9514 9611 ELSE
9515 9612 NODEPTR [DBG$B_PNARR_DTYPE] = DSC$K_DTYPE_BU
9516 9613 ELSE IF .STRIDE EQ 2
9517 9614 THEN
9518 9615 IF .SUBVECTOR[3]
9519 9616 THEN
9520 9617 NODEPTR [DBG$B_PNARR_DTYPE] = DSC$K_DTYPE_W
9521 9618 ELSE
9522 9619 NODEPTR [DBG$B_PNARR_DTYPE] = DSC$K_DTYPE_WU
9523 9620 ELSE IF .STRIDE EQ 4
9524 9621 THEN
9525 9622 IF .SUBVECTOR[3]
9526 9623 THEN
9527 9624 NODEPTR [DBG$B_PNARR_DTYPE] = DSC$K_DTYPE_L
9528 9625 ELSE
9529 9626 NODEPTR [DBG$B_PNARR_DTYPE] = DSC$K_DTYPE_LU
9530 9627 ELSE
9531 9628 BEGIN
9532 9629 IF .SUBVECTOR[3]
9533 9630 THEN
9534 9631 NODEPTR [DBG$B_PNARR_DTYPE] = DSC$K_DTYPE_V
9535 9632 ELSE
9536 9633 NODEPTR [DBG$B_PNARR_DTYPE] = DSC$K_DTYPE_VU;
9537 9634 NODEPTR[DBG$W_PNARR_LENGTH] = 8 * .NODEPTR[DBG$W_PNARR_LENGTH];
9538 9635 END;
9539 9636
9540 9637 ! Check that the block had exactly four subscripts.
9541 9638 !
9542 9639 IF .SUBSCR_COUNT LSS 4 THEN SIGNAL(DBG$TOOFEWSUB, 1, 4);
9543 9640 IF .SUBSCR_COUNT GTR 4 THEN SIGNAL(DBG$TOOMANSUB, 1, 4);
9544 9641
9545 9642 ! Check that the subscript values are in range.
9546 9643 !
9547 9644 IF NOT .REF_FLAG
9548 9645 THEN
9549 9646 IF (.SUBVECTOR[0] LSS 0) OR
9550 9647 (.SUBVECTOR[0] GEQ .DSTPTR[DST$L_BLI_BLOCK_UNITS])
9551 9648 THEN
9552 9649 SIGNAL(DBG$STRUCSIZE, 2,
9553 9650 .DSTPTR[DST$L_BLI_BLOCK_UNITS], .SUBVECTOR[0]);
9554 9651
9555 9652 IF (.SUBVECTOR[1] LSS -X'8000') OR
9556 9653 (.SUBVECTOR[1] GTR X'7FFF')
9557 9654 THEN
9558 9655 SIGNAL(DBG$_ILLOFFSET, 1, .SUBVECTOR[1]);
9559 9656
9560 9657
```

```

9561
9562
9563
9564
9565
9566
9567
9568
9569
9570
9571
9572
9573
9574
9575
9576
9577
9578
9579
9580
9581
9582
9583
9584
9585
9586
9587
9588
9589
9590
9591
9592
9593
9594
9595
9596
9597
9598
9599
9600
9601
9602
9603
9604
9605
9606
9607
9608
9609
9610
9611
9612
9613
9614
9615
9616
9617

```

```

IF (.SUBVECTOR[2] LSS 0)
THEN
    SIGNAL(DBG$_ILLLENGTH, 1, .SUBVECTOR[2]);

IF (.SUBVECTOR[2] GTR 32)
THEN
    BEGIN
        SUBVECTOR[2] = 32;
        SIGNAL(DBG$_SIZETRUNC);
    END;

IF (.SUBVECTOR[3] NEQ 0) AND (.SUBVECTOR[3] NEQ 1)
THEN
    SIGNAL(DBG$_ILLSIGEXT, 1, .SUBVECTOR[3]);

! Fill in the Primary Descriptor Sub-Node.
PRIMPTR [DBG$_DHDR_BLIBLK] = TRUE;
PRIMPTR [DBG$_DHDR_SUBREF] = TRUE;
PRIMPTR [DBG$_DHDR_BITREF] = TRUE;
PRIMPTR [DBG$_PRIM_OFFSET] = .SUBVECTOR[1];
PRIMPTR [DBG$_PRIM_LENGTH] = .SUBVECTOR[2];
PRIMPTR [DBG$_DHDR_SGNEXT] = .SUBVECTOR[3];
NODEPTR [DBG$_PNARR_SUBCNT] = 1;
NODESUBPTR [0, DBG$_PNSUB_SVALUE] = .SUBVECTOR[0];

END;                                ! End of BLISS block case

! Handle the Blockvector case.
[DST$_BLI_BLKVEC]:
BEGIN
    ! We previously represented the block as a block of longwords,
    ! for purposes of aggregate output. If we get here, however,
    ! we are no longer doing aggregate output.
    ! So, fix up the information
    ! here, filling in the correct stride.
    STRIDE = .DSTPTR [DST$_BLI_BLKVEC_UNIT_SIZE];
    NODESUBPTR [1, DBG$_PNSUB_STRIDE] = .STRIDE;
    NODESUBPTR [1, DBG$_PNSUB_UBOUND] =
        .DSTPTR [DST$_BLI_BLKVEC_UNITS]-1;
    NODEPTR [DBG$_PNARR_LENGTH] = .STRIDE;
    IF .STRIDE EQ 1
    THEN
        IF .SUBVECTOR[4]
        THEN
            NODEPTR [DBG$_PNARR_DTYPE] = DSC$_DTYPE_B
        ELSE
            NODEPTR [DBG$_PNARR_DTYPE] = DSC$_DTYPE_BU
    ELSE IF .STRIDE EQ 2
    THEN
        IF .SUBVECTOR[4]
        THEN

```



```
9618 9715
9619 9716
9620 9717
9621 9718
9622 9719
9623 9720
9624 9721
9625 9722
9626 9723
9627 9724
9628 9725
9629 9726
9630 9727
9631 9728
9632 9729
9633 9730
9634 9731
9635 9732
9636 9733
9637 9734
9638 9735
9639 9736
9640 9737
9641 9738
9642 9739
9643 9740
9644 9741
9645 9742
9646 9743
9647 9744
9648 9745
9649 9746
9650 9747
9651 9748
9652 9749
9653 9750
9654 9751
9655 9752
9656 9753
9657 9754
9658 9755
9659 9756
9660 9757
9661 9758
9662 9759
9663 9760
9664 9761
9665 9762
9666 9763
9667 9764
9668 9765
9669 9766
9670 9767
9671 9768
9672 9769
9673 9770
9674 9771

      NODEPTR [DBG$B_PNARR_DTYPE] = DSC$K_DTYPE_W
    ELSE
      NODEPTR [DBG$B_PNARR_DTYPE] = DSC$K_DTYPE_WU
    ELSE IF .STRIDE EQL 4
    THEN
      IF .SUBVECTOR[4]
      THEN
        NODEPTR [DBG$B_PNARR_DTYPE] = DSC$K_DTYPE_L
      ELSE
        NODEPTR [DBG$B_PNARR_DTYPE] = DSC$K_DTYPE_LU
    ELSE
      BEGIN
      IF .SUBVECTOR[4]
      THEN
        NODEPTR [DBG$B_PNARR_DTYPE] = DSC$K_DTYPE_V
      ELSE
        NODEPTR [DBG$B_PNARR_DTYPE] = DSC$K_DTYPE_VU;
        NODEPTR[DBG$W_PNARR_LENGTH] = 8 * .NODEPTR[DBG$W_PNARR_LENGTH];
      END;

      ! Check that the block-vector had exactly five subscripts.
      !
      IF .SUBSCR_COUNT LSS 5 THEN SIGNAL(DBG$TOOFEWSUB, 1, 5);
      IF .SUBSCR_COUNT GTR 5 THEN SIGNAL(DBG$TOOMANSUB, 1, 5);

      ! Check that the subscript values are in range.
      !
      IF NOT .REF_FLAG
      THEN
        BEGIN
        IF (.SUBVECTOR[0] LSS 0) OR
        (.SUBVECTOR[0] GEQ .DSTPTR[DST$L_BLI_BLKVEC_BLOCKS])
        THEN
          SIGNAL(DBG$STRUCSIZE, 2,
            .DSTPTR[DST$L_BLI_BLKVEC_BLOCKS], .SUBVECTOR[0]);

          IF (.SUBVECTOR[1] LSS 0) OR
          (.SUBVECTOR[1] GEQ .DSTPTR[DST$L_BLI_BLKVEC_UNITS])
          THEN
            SIGNAL(DBG$STRUCSIZE, 2,
              .DSTPTR[DST$L_BLI_BLKVEC_UNITS], .SUBVECTOR[1]);
          END;

          IF (.SUBVECTOR[2] LSS -X'8000') OR
          (.SUBVECTOR[2] GTR X'7FFF')
          THEN
            SIGNAL(DBG$_ILLOFFSET, 1, .SUBVECTOR[2]);

          IF (.SUBVECTOR[3] LSS 0)
          THEN
            SIGNAL(DBG$_ILLLENGTH, 1, .SUBVECTOR[3]);

          IF (.SUBVECTOR[3] GTR 32)
          THEN
            BEGIN
            SUBVECTOR[3] = 32;
```

```

9675      9772      4      SIGNAL(DBG$_SIZETRUNC);
9676      9773      END;
9677      9774
9678      9775      IF (.SUBVECTOR[4] NEQ 0) AND (.SUBVECTOR[4] NEQ 1)
9679      9776      THEN
9680      9777          SIGNAL(DBG$_ILLSIGEXT, 1, .SUBVECTOR[4]);
9681      9778
9682      9779      ! If a subscript range was specified, check that the lower range
9683      9780      ! value is also in range.
9684      9781
9685      9782      IF NOT .REF_FLAG
9686      9783      THEN
9687      9784          IF .NODEPTR[DBG$V_PNARR_RANGE] AND
9688      9785          ((.LOW_RANGE_VAL_LSS 0) OR
9689      9786          (.LOW_RANGE_VAL GEQ .DSTPTR[DST$_BLI_BLKVEC_BLOCKS]))
9690      9787          THEN
9691      9788              SIGNAL(DBG$_STRUCSIZE, 2,
9692      9789              .DSTPTR[DST$_BLI_BLKVEC_BLOCKS], .LOW_RANGE_VAL);
9693      9790
9694      9791
9695      9792      ! Fill in the Primary Descriptor Sub-Node.
9696      9793
9697      9794      PRIMPTR [DBG$V_DHDR_BLIBLK] = TRUE;
9698      9795      PRIMPTR [DBG$V_DHDR_SUBREF] = TRUE;
9699      9796      PRIMPTR [DBG$V_DHDR_BITREF] = TRUE;
9700      9797      PRIMPTR [DBG$V_PRIM_OFFSET] = .SUBVECTOR[2];
9701      9798      PRIMPTR [DBG$V_PRIM_LENGTH] = .SUBVECTOR[3];
9702      9799      PRIMPTR [DBG$V_DHDR_SGNEXT] = .SUBVECTOR[4];
9703      9800      NODEPTR [DBG$B_PNARR_SUBCNT] = 2;
9704      9801      NODESUBPTR [0, DBG$_PNSUB_SVALUE] = .SUBVECTOR[0];
9705      9802      NODESUBPTR [1, DBG$_PNSUB_SVALUE] = .SUBVECTOR[1];
9706      9803
9707      9804
9708      9805      ! If there was a range on the first subscript then make
9709      9806      ! the second one into a range too.
9710      9807
9711      9808      IF .NODEPTR[DBG$V_PNARR_RANGE]
9712      9809      THEN
9713      9810          BEGIN
9714      9811              NODESUBPTR [1, DBG$_PNSUB_LBOUND] = .SUBVECTOR[1];
9715      9812              NODESUBPTR [1, DBG$_PNSUB_UBOUND] = .SUBVECTOR[1];
9716      9813          END;
9717      9814      END;
9718      9815      ! End of BLISS block-vector case
9719      9816
9720      9817      ! Any other case should never occur and constitutes an internal
9721      9818      ! error in the BLISS Debug Symbol Table (DST).
9722      9819
9723      9820      [INRANGE, OTRANGE]:
9724      9821          SIGNAL(DBG$_INVDSTREC);
9725      9822
9726      9823      TES;
9727      9824      ! End of CASE of BLISS structure type
9728      9825
9729      9826      ! If a subscript range (as in ARR[2:5]) was specified for the first sub-
9730      9827      ! script, modify the array's lower and upper bounds in the Primary Descrip-
9731      9828      ! tor to represent the array "slice" specified by that subscript range.

```

```

: Build a new subnode. The typeid that we pass in to
: BUILD_PRIMARY_SUBNODE describes the element referenced by the
: subscript expression. This typeid is pulled from the CELLTYPE field.
:
DBG$BUILD_PRIMARY_SUBNODE (.PRIMPTR, RST$K_DATA, 0,
    RST$K_TYPE_ATOMIC, .NODEPTR[DBG$L_PNARR_CELLTYPE], 0);
RETURN;
END;

```

OFFC 00000 GET_BLISS SUBSCRIPTS:											
			5E		3C	C2	00002		WORD	Save R2,R3,R4,R5,R6,R7,R8,R9,R10,R11	9202
			56	04	AC	DO	00005		SUBL2	#60, SP	9275
		00000000G	00		56	DO	00009		MOVL	PRIMPTR, R6	
			52	0C	A6	DO	00010		MOVL	R6, DBG\$GL CURRENT_PRIMARY	9280
					12	12	00014		MOVL	12(R6), RSTPTR	9281
				08	AC	DD	00016		BNEQ	1\$	9283
					01	DD	00019		PUSHL	NAME	
					8F	DD	0001B		PUSHL	#1	
		00000000G	00	000281A8	03	FB	00021		PUSHL	#164264	
			54	0C	A2	DO	00028	1\$:	CALLS	#3, LIB\$SIGNAL	9284
			AE	04	A6	9E	0002C		MOVL	12(RSTPTR), DSTPTR	9285
06	04	BE	08		18	ED	00031		MOVAB	4(R6), 4(SP)	
					12	13	00037		CMPZV	#24, #8, 24(SP), #6	
				07	A4	9F	00039		BEQL	2\$	
					01	DD	0003C		PUSHAB	7(DSTPTR)	9287
					8F	DD	0003E		PUSHL	#1	
		00000000G	00	000281A8	03	FB	00044		PUSHL	#164264	
				1C	AE	9F	0004B	2\$:	CALLS	#3, LIB\$SIGNAL	9289
				24	AE	9F	0004E		PUSHAB	TYPEID	
					52	DD	00051		PUSHAB	FCODE	
		00000000G	00		03	FB	00053		PUSHL	RSTPTR	
			0D	20	AE	D1	0005A		CALLS	#3, DBG\$STA_SYMTYPE	9290
					12	13	0005E		CMPL	FCODE, #13	
				07	A4	9F	00060		BEQL	3\$	
					01	DD	00063		PUSHAB	7(DSTPTR)	9292
					8F	DD	00065		PUSHL	#1	
		00000000G	00	000281A8	03	FB	0006B		PUSHL	#164264	
			03		00	EF	00072	3\$:	CALLS	#3, LIB\$SIGNAL	
58	05	A4			12	12	00078		EXTZV	#0, #3, 5(DSTPTR), STRUC	9300
					12	12	00078		BNEQ	4\$	9301
				07	A4	9F	0007A		PUSHAB	7(DSTPTR)	9303

		000281A8	01	DD	0007D	PUSHL	#1		
			8F	DD	0007F	PUSHL	#164264		
00000000G	00		03	FB	00085	CALLS	#3, LIB\$SIGNAL		
		05	A4	95	0008C	48:	TSTB	5(DSTPTR)	9310
			1E	18	0008F	BGEQ	5\$		
			54	DD	00091	PUSHL	DSTPTR		9314
		08	A6	DD	00093	PUSHL	8(R6)		
			0D	DD	00096	PUSHL	#13		9313
	7E		06	7D	00098	MOVQ	#6, -(SP)		
			56	DD	0009B	PUSHL	R6		
D88C	CF		06	FB	0009D	CALLS	#6, DBG\$BUILD_PRIMARY_SUBNODE		
	52	14	A6	D0	000A2	MOVL	20(R6), NODEPTR		9315
0A	A2		01	88	000A6	BISB2	#1, 10(NODEPTR)		9316
	6E		01	D0	000AA	MOVL	#1, REF_FLAG		9317
			02	11	000AD	BRB	6\$		9310
			6E	D4	000AF	5\$:	CLRL	REF_FLAG	9320
	52	18	A6	D0	000B1	6\$:	MOVL	24(R6), NODEPTR	9326
	53	28	A2	9E	000B5	MOVAB	40(R2), NODESUBPTR		9327
	5B	08	A2	9E	000B9	MOVAB	8(NODEPTR), R11		9328
02	AB		01	88	000BD	BISB2	#1, 2(R11)		
			55	D4	000C1	CLRL	SUBSCR_COUNT		9336
00000000'	EF		01	D0	000C3	MOVL	#1, TERMINATOR_CODE		9337
	02	00000000'	EF	D1	000CA	7\$:	CMPL	TERMINATOR_CODE, #2	9338
			03	12	000D1	BNEQ	8\$		
			01A1	31	000D3	BRW	29\$		
	59	00000000'	EF	D0	000D6	8\$:	MOVL	CHARPTR, LA_PTR	9346
	20		69	91	000DD	9\$:	CMPB	(LA_PTR), #32	9347
			04	12	000E0	BNEQ	10\$		
			59	D6	000E2	INCL	LA_PTR		
			F7	11	000E4	BRB	9\$		
	2A		69	91	000E6	10\$:	CMPB	(LA_PTR), #42	9348
			03	13	000E9	BEQL	11\$		
			00B2	31	000EB	BRW	17\$		
13	6B		13	E0	000EE	11\$:	BBS	#19, (R11), 12\$	9351
			55	D5	000F2	TSTL	SUBSCR_COUNT		9352
			0F	12	000F4	BNEQ	12\$		
	02		5B	D1	000F6	CMPL	STRUC, #2		9353
			17	13	000F9	BEQL	13\$		
	01		5B	D1	000FB	CMPL	STRUC, #1		9354
			12	13	000FE	BEQL	13\$		
	04		5B	D1	00100	CMPL	STRUC, #4		9355
			0D	13	00103	BEQL	13\$		
		00028F08	8F	DD	00105	12\$:	PUSHL	#167688	9357
00000000G	00		01	FB	0010B	CALLS	#1, LIB\$SIGNAL		
00000000'	EF	01	A9	9E	00112	13\$:	MOVAB	1(R9), CHARPTR	9359
			55	D6	0011A	INCL	SUBSCR_COUNT		9360
			7E	D4	0011C	CLRL	-(SP)		9368
		00000000'	EF	DD	0011E	PUSHL	SUBSCRIPT_TERM_TBL		9369
			7E	7C	00124	CLRQ	-(SP)		9368
E235	CF		04	FB	00126	CALLS	#4, DBG\$LEXICAL_SCANNER		
18	AE		50	D0	0012B	MOVL	R0, TOKEN		
	50	00000000'	EF	9E	0012F	MOVAB	TERMINATOR_TOKEN, R0		9370
	50	18	AE	D1	00136	CMPL	TOKEN, R0		
			1E	13	0013A	BEQL	14\$		
	24	AE	01	90	0013C	MOVAB	#1, ASCII_STRING		9375
	25	AE	FF	90	00140	MOVAB	@CHARPTR, ASCII_STRING+1		9376
		24	AE	9F	00148	PUSHAB	ASCII_STRING		9377

			01	DD	0014B	PUSHL	#1		
			8F	DD	0014D	PUSHL	#166370		
00000000G	00		03	FB	00153	CALLS	#3, LIBSSIGNAL		
	03	00000000'	EF	D1	0015A	14%: CMPL	TERMINATOR_CODE, #3		9379
			0D	12	00161	BNEQ	15%		
			8F	DD	00163	PUSHL	#167688		9381
00000000G	00	00028F08	01	FB	00169	CALLS	#1, LIBSSIGNAL		
		00000000'	EF	D5	00170	15%: TSTL	TERMINATOR_CODE		9382
			0D	12	00176	BNEQ	16%		
			8F	DD	00178	PUSHL	#167568		9384
00000000G	00	00028E90	01	FB	0017E	CALLS	#1, LIBSSIGNAL		
00000000'	EF	00000000'	EF	C0	00185	16%: ADDL2	TERMINATOR_LENGTH, CHARPTR		9385
02	AB		08	88	00190	BISB2	#8, 2(R11)		9390
	5A	08	A3	D0	00194	MOVL	8(NODESUBPTR), LOW RANGE VAL		9391
28	AE	0C	A3	D0	00198	MOVL	12(NODESUBPTR), SUBVECTOR		9392
			FF	2A	31	0019D	BRW	7%	9348
10	AE	00000000'	EF	D0	001A0	17%: MOVL	EXPRESSION_RADIX, SAVED_RADIX		9404
00000000'	EF		0A	D0	001A8	MOVL	#10, EXPRESSION_RADIX		9405
		00000000'	EF	DD	001AF	PUSHL	SUBSCRIPT_TERM_TBL		9406
			7E	D4	001B5	CLRL	-(SP)		
DCFE	CF		02	FB	001B7	CALLS	#2, DBGSEXPRESSION_PARSER		
	57		50	D0	001BC	MOVL	R0, VALPTR		
00000000'	EF	10	AE	D0	001BF	MOVL	SAVED_RADIX, EXPRESSION_RADIX		9407
		00000000'	EF	D5	001C7	TSTL	TERMINATOR_CODE		9414
			0D	12	001CD	BNEQ	18%		
			8F	DD	001CF	PUSHL	#167568		
00000000G	00	00028E90	01	FB	001D5	CALLS	#1, LIBSSIGNAL		
00000000'	EF	00000000'	EF	C0	001DC	18%: ADDL2	TERMINATOR_LENGTH, CHARPTR		9415
	0E	06	A7	91	001E7	CMPB	6(VALPTR), #14		9422
			38	12	001EB	BNEQ	23%		
11	6B		13	E1	001ED	BBC	#19, (R11), 19%		9425
			55	D5	001F1	TSTL	SUBSCR_COUNT		
			0D	12	001F3	BNEQ	19%		
			8F	DD	001F5	PUSHL	#167688		9427
00000000G	00	00028F08	01	FB	001FB	CALLS	#1, LIBSSIGNAL		
08	AE	20	A7	9E	00202	19%: MOVAB	32(R7), PTR		9429
14	AE	08	BE	D0	00207	MOVL	@PTR, COUNT		9430
			50	D4	0020C	CLRL	I		9431
			0E	11	0020E	BRB	22%		
	05		55	D1	00210	20%: CMPL	SUBSCR_COUNT, #5		9433
			07	18	00213	BGEQ	21%		
28	AE45	08	BE40	D0	00215	MOVL	@PTR[I], SUBVECTOR[SUBSCR_COUNT]		
			55	D6	0021C	21%: INCL	SUBSCR_COUNT		9434
ED	50	14	AE	F3	0021E	22%: AOBLEQ	COUNT, -1, 20%		9431
			4F	11	00223	BRB	28%		9422
			57	DD	00225	23%: PUSHL	VALPTR		9450
FB33	CF		01	FB	00227	CALLS	#1, CONVERT_TO_INTEGER		
0C	AE		50	D0	0022C	MOVL	R0, VALUE		
	03	00000000'	EF	D1	00230	CMPL	TERMINATOR_CODE, #3		9459
			2E	12	00237	BNEQ	26%		
13	6B		13	E0	00239	BBS	#19, (R11), 24%		9462
			55	D5	0023D	TSTL	SUBSCR_COUNT		9463
			0F	12	0023F	BNEQ	24%		
	02		58	D1	00241	CMPL	STRUC, #2		9464
			17	13	00244	BEQL	25%		
	01		58	D1	00246	CMPL	STRUC, #1		9465
			12	13	00249	BEQL	25%		

	04		58	D1	0024B		CMPL	STRUC, #4	9466
			0D	13	0024E		BEQL	25\$	
		00028F08	8F	DD	00250	24\$:	PUSHL	#167688	9468
	00		01	FB	00256		CALLS	#1, LIB\$SIGNAL	
	02		08	88	0025D	25\$:	BISB2	#8, 2(R11)	9470
	SA	OC	AE	D0	00261		MOVL	VALUE, LOW_RANGE_VAL	9471
			0D	11	00265		BRB	28\$	9459
	05		55	D1	00267	26\$:	CMPL	SUBSCR_COUNT, #5	9483
			06	18	0026A		BGEQ	27\$	
	28	AE45	OC	AE	D0	0026C	MOVL	VALUE, SUBVECTOR[SUBSCR_COUNT]	9485
			55	D6	00272	27\$:	INCL	SUBSCR_COUNT	9487
			FE53	31	00274	28\$:	BRW	7\$	9338
	04	00	58	CF	00277	29\$:	CASEL	STRUC, #0, #4	9500
00A9	001A	0036	000A		0027B	30\$:	.WORD	31\$-30\$,-	
			01F7		00283			34\$-30\$,-	
								32\$-30\$,-	
								42\$-30\$,-	
								62\$-30\$,-	
		0002832A	8F	DD	00285	31\$:	PUSHL	#164650	9821
	00		01	FB	0028B		CALLS	#1, LIB\$SIGNAL	
			037E	31	00292		BRW	85\$	
			55	D5	00295	32\$:	TSTL	SUBSCR_COUNT	9512
			11	14	00297		BGTR	33\$	
			01	DD	00299		PUSHL	#1	
			01	DD	0029B		PUSHL	#1	
		00028EA0	8F	DD	0029D		PUSHL	#167584	
	00		03	FB	002A3		CALLS	#3, LIB\$SIGNAL	
	01		55	D1	002AA	33\$:	CMPL	SUBSCR_COUNT, #1	9513
			1C	14	002AD		BGTR	36\$	
			2B	11	002AF		BRB	37\$	9518
			55	D5	002B1	34\$:	TSTL	SUBSCR_COUNT	9556
			11	14	002B3		BGTR	35\$	
			01	DD	002B5		PUSHL	#1	
			01	DD	002B7		PUSHL	#1	
		00028EA0	8F	DD	002B9		PUSHL	#167584	
	00		03	FB	002BF		CALLS	#3, LIB\$SIGNAL	
	01		55	D1	002C6	35\$:	CMPL	SUBSCR_COUNT, #1	9557
			11	15	002C9		BLEQ	37\$	
			01	DD	002CB	36\$:	PUSHL	#1	
			01	DD	002CD		PUSHL	#1	
		00028EB0	8F	DD	002CF		PUSHL	#167600	
	00		03	FB	002D5		CALLS	#3, LIB\$SIGNAL	
	42		6E	EB	002DC	37\$:	BLBS	REF FLAG, 41\$	9562
	50	28	AE	D0	002DF		MOVL	SUBVECTOR, R0	9565
			06	19	002E3		BLSS	38\$	
	06	A4	50	D1	002E5		CMPL	R0, 6(DSTPTR)	9566
			14	19	002E9		BLSS	39\$	
			50	DD	002EB	38\$:	PUSHL	R0	9569
		06	A4	DD	002ED		PUSHL	6(DSTPTR)	
			02	DD	002F0		PUSHL	#2	9568
		0002807B	8F	DD	002F2		PUSHL	#163963	
	00		04	FB	002F8		CALLS	#4, LIB\$SIGNAL	
1E	68		13	E1	002FF	39\$:	BBC	#19, (R11), 41\$	9575
			5A	D5	00303		TSTL	LOW_RANGE_VAL	9576
			06	19	00305		BLSS	40\$	
	06	A4	5A	D1	00307		CMPL	LOW_RANGE_VAL, 6(DSTPTR)	9577
			14	19	0030B		BLSS	41\$	

50	0A	A4	04	04	06	5A	DD	0030D	40\$:	PUSHL	LOW_RANGE_VAL	9580
			06	A3	02	A4	DD	0030F		PUSHL	6(DSTPTR)	
			1C	A4	02	8F	DD	00312		PUSHL	#2	9579
				A2	04	04	FB	00314		PUSHL	#163963	
				51	01	43	31	00321	41\$:	CALLS	#4, LIB\$SIGNAL	
				01	00	00	EF	00324	42\$:	BRW	61\$	9586
				18	50	00	DD	0032A		EXTZV	#0, #4, 10(DSTPTR), STRIDE	9601
				01	01	50	DD	0032E		MOVL	STRIDE, 4(NODESUBPTR)	9602
				51	50	01	C3	00334		SUBL3	#1, 6(DSTPTR), 12(NODESUBPTR)	9604
				01	50	01	B0	00338		MOVW	STRIDE, 28(NODEPTR)	9605
				01	50	18	A2	9E		MOVAB	24(NODEPTR), R1	9610
				01	50		D1	0033C		CMPL	STRIDE, #1	9606
				06	10		12	0033F		BNEQ	44\$	
				02	34	AE	E9	00341		BLBC	SUBVECTOR+12, 43\$	9608
				02	06	06	90	00345		MOVB	#6, 2(R1)	9610
				02	42	11	00349		BRB	51\$		
				02	02	02	90	0034B	43\$:	MOVB	#2, 2(R1)	9612
				02	3C	11	0034F		BRB	51\$		9608
				02	50	D1	00351		CMPL	STRIDE, #2		9613
				06	10	12	00354		BNEQ	46\$		
				02	34	AE	E9	00356		BLBC	SUBVECTOR+12, 45\$	9615
				02	07	90	0035A		MOVB	#7, 2(R1)		9617
				02	2D	11	0035E		BRB	51\$		
				02	03	90	00360		MOVB	#3, 2(R1)		9619
				04	27	11	00364		BRB	51\$		9615
				04	50	D1	00366		CMPL	STRIDE, #4		9620
				06	10	12	00369		BNEQ	48\$		
				02	34	AE	E9	0036B		BLBC	SUBVECTOR+12, 47\$	9622
				02	08	90	0036F		MOVB	#8, 2(R1)		9624
				02	18	11	00373		BRB	51\$		
				02	04	90	00375		MOVB	#4, 2(R1)		9626
				02	12	11	00379		BRB	51\$		9622
				06	34	AE	E9	0037B	48\$:	BLBC	SUBVECTOR+12, 49\$	9629
				02	01	90	0037F		MOVB	#1, 2(R1)		9631
				02	04	11	00383		BRB	50\$		
				02	22	90	00385		MOVB	#34, 2(R1)		9633
				1C	08	A4	00389		MULW2	#8, 28(NODEPTR)		9634
				04	55	D1	0038D		CMPL	SUBSCR_COUNT, #4		9639
					11	18	00390		BGEQ	52\$		
					04	DD	00392		PUSHL	#4		
					01	DD	00394		PUSHL	#1		
					8F	DD	00396		PUSHL	#167584		
					03	FB	0039C		CALLS	#3, LIB\$SIGNAL		
					55	D1	003A3		CMPL	SUBSCR_COUNT, #4		9640
					11	15	003A6		BLEQ	53\$		
					04	DD	003A8		PUSHL	#4		
					01	DD	003AA		PUSHL	#1		
					8F	DD	003AC		PUSHL	#167600		
					03	FB	003B2		CALLS	#3, LIB\$SIGNAL		
					6E	E8	003B9		BLBS	REF_FLAG, 55\$		9645
					50	AE	DD	003BC	53\$:	MOVL	SUBVECTOR, R0	9647
					06	19	003C0		BLSS	54\$		
					50	D1	003C2		CMPL	R0, 6(DSTPTR)		9648
					14	19	003C6		BLSS	55\$		
					50	DD	003C8		PUSHL	R0		9651
					06	A4	DD	003CA	54\$:	PUSHL	6(DSTPTR)	
					02	DD	003CD		PUSHL	#2		9650

00000000G	00	0002807B	8F	DD	003CF	PUSHL	#163963	
FFFF8000	8F	2C	04	FB	003D5	CALLS	#4, LIB\$SIGNAL	9653
			AE	D1	003DC	CMPL	SUBVECTOR+4, #-32768	
00007FFF	8F	2C	0A	19	003E4	BLSS	56\$	9654
			AE	D1	003E6	CMPL	SUBVECTOR+4, #32767	
		2C	12	15	003EE	BLEQ	57\$	9656
			AE	DD	003F0	PUSHL	SUBVECTOR+4	
			01	DD	003F3	PUSHL	#1	
00000000G	00	000280F0	8F	DD	003F5	PUSHL	#164080	
	58	30	03	FB	003FB	CALLS	#3, LIB\$SIGNAL	9658
			AE	D0	00402	MOVL	SUBVECTOR+8, R8	
			11	18	00406	BGEQ	58\$	9660
			58	DD	00408	PUSHL	R8	
			01	DD	0040A	PUSHL	#1	
00000000G	00	00028EE8	8F	DD	0040C	PUSHL	#167656	
	20		03	FB	00412	CALLS	#3, LIB\$SIGNAL	9662
			58	D1	00419	CMPL	R8, #32	
			11	15	0041C	BLEQ	59\$	
30	AE		20	D0	0041E	MOVL	#32, SUBVECTOR+8	9665
00000000G	00	00028073	8F	DD	00422	PUSHL	#163955	9666
	58	34	01	FB	00428	CALLS	#1, LIB\$SIGNAL	
			AE	D0	0042F	MOVL	SUBVECTOR+12, R8	9669
			16	13	00433	BEQL	60\$	
	01		58	D1	00435	CMPL	R8, #1	
			11	13	00438	BEQL	60\$	
			58	DD	0043A	PUSHL	R8	9671
			01	DD	0043C	PUSHL	#1	
00000000G	00	00028140	8F	DD	0043E	PUSHL	#164160	
	04		03	FB	00444	CALLS	#3, LIB\$SIGNAL	9676
	04		10	88	0044B	BISB2	#16, @4(SP)	9677
	04		02	88	0044F	BISB2	#2, @4(SP)	9678
	04		04	88	00453	BISB2	#4, @4(SP)	9679
	10	2C	AE	B0	00457	MOVW	SUBVECTOR+4, 16(R6)	9680
04	BE		AE	B0	0045C	MOVW	SUBVECTOR+8, 18(R6)	9681
			58	F0	00461	INSV	R8, #3, #1, @4(SP)	9682
	12	30	01	90	00467	MOVB	#1, 31(NODEPTR)	9683
	03		AE	D0	0046B	MOVL	SUBVECTOR, (NODESUBPTR)	9500
	1F		01A1	31	0046F	BRW	85\$	9699
	63	28	A4	9A	00472	MOVZBL	14(DSTPTR), STRIDE	9700
		0E	50	D0	00476	MOVL	STRIDE, 24(NODESUBPTR)	9702
20	A3		01	C3	0047A	SUBL3	#1, 10(DSTPTR), 32(NODESUBPTR)	9703
			50	B0	00480	MOVW	STRIDE, 28(NODEPTR)	9708
		18	A2	9E	00484	MOVAB	24(NODEPTR), R1	9704
			50	D1	00488	CMPL	STRIDE, #1	
			10	12	0048B	BNEQ	64\$	
	06	38	AE	E9	0048D	BLBC	SUBVECTOR+16, 63\$	9706
	02		06	90	00491	MOVB	#6, 2(R1)	9708
			42	11	00495	BRB	71\$	
	02		02	90	00497	MOVB	#2, 2(R1)	9710
			3C	11	0049B	BRB	71\$	9706
	02		50	D1	0049D	CMPL	STRIDE, #2	9711
			10	12	004A0	BNEQ	66\$	
	06	38	AE	E9	004A2	BLBC	SUBVECTOR+16, 65\$	9713
	02		07	90	004A6	MOVB	#7, 2(R1)	9715
			2D	11	004AA	BRB	71\$	
	02		03	90	004AC	MOVB	#3, 2(R1)	9717
			27	11	004B0	BRB	71\$	9713

	04		50	D1	004B2	66\$:	CMPL	STRIDE, #4	9718
			10	12	004B5		BNEQ	68\$	
	06	38	AE	E9	004B7		BLBC	SUBVECTOR+16, 67\$	9720
02	A1		08	90	004BB		MOVB	#8, 2(R1)	9722
			18	11	004BF		BRB	71\$	
02	A1		04	90	004C1	67\$:	MOVB	#4, 2(R1)	9724
			12	11	004C5		BRB	71\$	9720
	06	38	AE	E9	004C7	68\$:	BLBC	SUBVECTOR+16, 69\$	9727
02	A1		01	90	004CB		MOVB	#1, 2(R1)	9729
			04	11	004CF		BRB	70\$	
02	A1		22	90	004D1	69\$:	MOVB	#34, 2(R1)	9731
1C	A2		08	A4	004D5	70\$:	MULW2	#8, 28(NODEPTR)	9732
	05		55	D1	004D9	71\$:	CMPL	SUBSCR_COUNT, #5	9737
			11	18	004DC		BGEQ	72\$	
			05	DD	004DE		PUSHL	#5	
			01	DD	004E0		PUSHL	#1	
00000000G	00	00028EA0	8F	DD	004E2		PUSHL	#167584	
	05		03	FB	004E8		CALLS	#3, LIB\$SIGNAL	
			55	D1	004EF	72\$:	CMPL	SUBSCR_COUNT, #5	9738
			11	15	004F2		BLEQ	73\$	
			05	DD	004F4		PUSHL	#5	
			01	DD	004F6		PUSHL	#1	
00000000G	00	00028EB0	8F	DD	004F8		PUSHL	#167600	
	40		03	FB	004FE		CALLS	#3, LIB\$SIGNAL	
	50		6E	E8	00505	73\$:	BLBS	REF FLAG, 77\$	9743
		28	AE	D0	00508		MOVL	SUBVECTOR, R0	9746
			06	19	0050C		BLSS	74\$	
06	A4		50	D1	0050E		CMPL	R0, 6(DSTPTR)	9747
			14	19	00512		BLSS	75\$	
			50	DD	00514	74\$:	PUSHL	R0	9750
		06	A4	DD	00516		PUSHL	6(DSTPTR)	
			02	DD	00519		PUSHL	#2	9749
		0002807B	8F	DD	0051B		PUSHL	#163963	
00000000G	00		04	FB	00521		CALLS	#4, LIB\$SIGNAL	
	50	2C	AE	D0	00528	75\$:	MOVL	SUBVECTOR+4, R0	9752
			06	19	0052C		BLSS	76\$	
0A	A4		50	D1	0052E		CMPL	R0, 10(DSTPTR)	9753
			14	19	00532		BLSS	77\$	
			50	DD	00534	76\$:	PUSHL	R0	9756
		0A	A4	DD	00536		PUSHL	10(DSTPTR)	
			02	DD	00539		PUSHL	#2	9755
		0002807B	8F	DD	0053B		PUSHL	#163963	
00000000G	00		04	FB	00541		CALLS	#4, LIB\$SIGNAL	
FFFF8000	8F	30	AE	D1	00548	77\$:	CMPL	SUBVECTOR+8, #-32768	9759
			0A	19	00550		BLSS	78\$	
00007FFF	8F	30	AE	D1	00552		CMPL	SUBVECTOR+8, #32767	9760
			12	15	0055A		BLEQ	79\$	
		30	AE	DD	0055C	78\$:	PUSHL	SUBVECTOR+8	9762
			01	DD	0055F		PUSHL	#1	
		000280F0	8F	DD	00561		PUSHL	#164080	
00000000G	00		03	FB	00567		CALLS	#3, LIB\$SIGNAL	
	58	34	AE	D0	0056E	79\$:	MOVL	SUBVECTOR+12, R8	9764
			11	18	00572		BGEQ	80\$	
			58	DD	00574		PUSHL	R8	9766
			01	DD	00576		PUSHL	#1	
		00028EE8	8F	DD	00578		PUSHL	#167656	
00000000G	00		03	FB	0057E		CALLS	#3, LIB\$SIGNAL	

		20		58	D1	00585	80\$:	CMPL	R8, #32	9768
				11	15	00588		BLEQ	81\$	
	34	AE		20	DD	0058A		MOVL	#32, SUBVECTOR+12	9771
			00028073	8F	DD	0058E		PUSHL	#163955	9772
	00000000G	00		01	FB	00594		CALLS	#1, LIB\$SIGNAL	
		58	38	AE	DD	0059B	81\$:	MOVL	SUBVECTOR+16, R8	9775
				16	13	0059F		BEQL	82\$	
		01		58	D1	005A1		CMPL	R8, #1	
				11	13	005A4		BEQL	82\$	
				58	DD	005A6		PUSHL	R8	9777
				01	DD	005AB		PUSHL	#1	
			00028140	8F	DD	005AA		PUSHL	#164160	
	00000000G	00		03	FB	005B0		CALLS	#3, LIB\$SIGNAL	
		22		6E	E8	005B7	82\$:	BLBS	REF_FLAG, 84\$	9783
1E		6B		13	E1	005BA		BBC	#19, (R11), 84\$	9785
				5A	D5	005BE		TSTL	LOW_RANGE_VAL	9786
				06	19	005C0		BLSS	83\$	
	06	A4		5A	D1	005C2		CMPL	LOW_RANGE_VAL, 6(DSTPTR)	9787
				14	19	005C6		BLSS	84\$	
			06	5A	DD	005C8	83\$:	PUSHL	LOW_RANGE_VAL	9790
				A4	DD	005CA		PUSHL	6(DSTPTR)	
				02	DD	005CD		PUSHL	#2	9789
			0002807B	8F	DD	005CF		PUSHL	#163963	
	00000000G	00		04	FB	005D5		CALLS	#4, LIB\$SIGNAL	
		04		10	88	005DC	84\$:	BISB2	#16, @4(SP)	9795
		04		02	88	005E0		BISB2	#2, @4(SP)	9796
		04		04	88	005E4		BISB2	#4, @4(SP)	9797
		10	A6	30	AE	B0	005E8	MOVW	SUBVECTOR+8, 16(R6)	9798
		12	A6	34	AE	B0	005ED	MOVW	SUBVECTOR+12, 18(R6)	9799
04	BE			58	F0	005F2		INSV	R8, #3, #1, @4(SP)	9800
		01		02	90	005F8		MOVB	#2, 31(NODEPTR)	9801
		1F	A2		AE	DD	005FC	MOVL	SUBVECTOR, (NODESUBPTR)	9802
		63		28	AE	DD	00600	MOVL	SUBVECTOR+4, 20(NODESUBPTR)	9803
		14	A3	2C	AE	DD	00605	BBC	#19, (R11), 87\$	9808
				13	E1	00609		MOVL	SUBVECTOR+4, 28(NODESUBPTR)	9811
	2E			AE	DD	0060E		MOVL	SUBVECTOR+4, 32(NODESUBPTR)	9812
		1C	A3	2C	AE	DD	00613	BBC	#19, (R11), 87\$	9830
		20	A3	2C	AE	DD	00617	CMPL	LOW_RANGE_VAL, SUBVECTOR	9833
		28	6B		0D	15	0061B	BLEQ	86\$	
				8F	DD	0061D		PUSHL	#167688	
			00028F08	01	FB	00623		CALLS	#1, LIB\$SIGNAL	
	00000000G	00		5A	DD	0062A	86\$:	MOVL	LOW_RANGE_VAL, (NODESUBPTR)	9834
		63		5A	DD	0062D		MOVL	LOW_RANGE_VAL, 8(NODESUBPTR)	9835
		08	A3		AE	DD	00631	MOVL	SUBVECTOR, 12(NODESUBPTR)	9836
		0C	A3	28	AE	DD	00636	RET		9832
				7E	D4	00637	87\$:	CLRL	-(SP)	9845
				A2	DD	00639		PUSHL	36(NODEPTR)	9846
				02	DD	0063C		PUSHL	#2	9845
		7E		06	7D	0063E		MOVQ	#6, -(SP)	
				56	DD	00641		PUSHL	R6	
	D2E6	CF		06	FB	00643		CALLS	#6, DBG\$BUILD_PRIMARY_SUBNODE	
				04	00648			RET		9848

; Routine Size: 1609 bytes, Routine Base: DBG\$CODE + 273E

```
9753 9849 1 ROUTINE GET_DEREFERENCE (PRIMPTR): NOVALUE =
9754 9850 1
9755 9851 1 FUNCTION
9756 9852 1     This routine is called upon seeing the dereference operator, e.g.,
9757 9853 1     the ^ in a PASCAL primary such as A^.
9758 9854 1
9759 9855 1     If the object being dereferenced is a pointer or a file variable, then
9760 9856 1     this routine lights a bit in the current primary subnode which
9761 9857 1     indicates that the dereference is taking place. It then calls
9762 9858 1     DBG$BUILD_PRIMARY_SUBNODE to append a new subnode. The type
9763 9859 1     information in the new subnode reflects the type of the object
9764 9860 1     being pointed to; or in the case of file variables, the type
9765 9861 1     of the objects in the file.
9766 9862 1
9767 9863 1 INPUTS
9768 9864 1     PRIMPTR          - A pointer to the Primary Descriptor currently
9769 9865 1                      being constructed by DBG$PRIMARY_PARSER.
9770 9866 1
9771 9867 1 OUTPUTS
9772 9868 1     The Primary Descriptor pointed to by PRIMPTR is modified.
9773 9869 1
9774 9870 1 BEGIN
9775 9871 1
9776 9872 1 MAP
9777 9873 1     PRIMPTR: REF DBG$PRIMARY;
9778 9874 1
9779 9875 1 LOCAL
9780 9876 1     FCODE,                ! Local variable holding fcode info
9781 9877 1     JUNK,                 ! Dummy output parameter
9782 9878 1     NODEPTR: REF DBG$PRIM_NODE, ! Points to a Primary Sub-node
9783 9879 1     TYPEID;               ! Pointer to a RST type entry
9784 9880 1
9785 9881 1 DBG$GL_CURRENT_PRIMARY = .PRIMPTR;
9786 9882 1
9787 9883 1 ! Check that the object being dereferenced is actually a pointer.
9788 9884 1 !
9789 9885 1 !
9790 9886 1 IF .PRIMPTR[DBG$B_DHDR_FCODE] NEQ RST$K_TYPE_TPTR
9791 9887 1 AND .PRIMPTR[DBG$B_DHDR_FCODE] NEQ RST$K_TYPE_FILE
9792 9888 1 THEN
9793 9889 1     SIGNAL(DBG$_NOTPTR);
9794 9890 1
9795 9891 1
9796 9892 1 ! Obtain a pointer to the bottom level sub-node by following the
9797 9893 1 ! back-pointer. Light the EVAL bit in this subnode,
9798 9894 1 ! which indicates that pointer dereferencing is
9799 9895 1 ! taking place.
9800 9896 1 ! Then, obtain the pointer to the RST type entry for the object being
9801 9897 1 ! dereferenced.
9802 9898 1
9803 9899 1 NODEPTR = .PRIMPTR [DBG$L PRIM_BLINK];
9804 9900 1 NODEPTR [DBG$V_PNODE_EVAL] = TRUE;
9805 9901 1 TYPEID = .NODEPTR [DBG$L_PNODE_TYPEID];
9806 9902 1
9807 9903 1
9808 9904 1 ! From this typeid, get the typeid for the object being pointed to.
9809 9905 1 ! For pointer variables, use the routine that extracts the typeid
```

```
9810 9906 2 | of the pointed-to object.
9811 9907 | For file variables, use the routine that extracts the typeid of
9812 9908 | objects in the file.
9813 9909 | Then obtain the fcode from the typeid.
9814 9910 |
9815 9911 | IF .PRIMPTR(DBG$B_DHDR_FCODE) EQL RST$K_TYPE_TPTR
9816 9912 | THEN
9817 9913 |     DBG$STA_TYP_TYPEDPTR (.TYPEID, TYPEID)
9818 9914 | ELSE
9819 9915 |     DBG$STA_TYP_FILE (.TYPEID, JUNK, TYPEID);
9820 9916 | FCODE = DBG$STA_TYPEFCODE (.TYPEID);
9821 9917 |
9822 9918 |
9823 9919 | Append a new sub-node to the Primary Descriptor.
9824 9920 |
9825 9921 | DBG$BUILD_PRIMARY_SUBNODE (.PRIMPTR, RST$K_DATA, 0, .FCODE, .TYPEID, 0);
9826 9922 | RETURN;
9827 9923 | END;
```

```
0004 00000 GET_DEREFERENCE:
                                .WORD Save R2
                                SUBL2 #8, SP
                                MOVL PRIMPTR, R2
00000000G 00 04 AC D0 00005 MOVL R2, DBG$GL_CURRENT_PRIMARY
00 06 A2 91 00010 CMPB 6(R2), #6
0F 06 A2 91 00016 BEQL 1$
0D 13 00014 BEQL 1$
00000000G 00 06 A2 91 00016 BEQL 1$
0A 50 18 A2 D0 00029 1$: CALLS #1, LIB$SIGNAL
0A A0 0C 01 88 0002D MOVL 24(R2), NODEPTR
06 06 A0 D0 00031 BISB2 #1, 10(NODEPTR)
06 06 A2 91 00035 MOVL 12(NODEPTR), TYPEID
0E 12 00039 CMPB 6(R2), #6
0E DD 0003B BNEQ 2$
04 AE DD 0003D PUSHL SP
00 04 AE DD 0003D PUSHL TYPEID
02 FB 00040 CALLS #2, DBG$STA_TYP_TYPEDPTR
0F 11 00047 BRB 3$
08 SE DD 00049 2$: PUSHL SP
08 AE 9F 0004B PUSHAB JUNK
08 AE DD 0004E PUSHL TYPEID
00000000G 00 08 03 FB 00051 CALLS #3, DBG$STA_TYP_FILE
06 DD 00058 3$: PUSHL TYPEID
00000000G 00 01 FB 0005A CALLS #1, DBG$STA_TYPEFCODE
07E D4 00061 CLRL -(SP)
04 AE DD 00063 PUSHL TYPEID
05 DD 00066 PUSHL FCODE
07E 06 7D 00068 MOVQ #6, -(SP)
05 DD 00068 PUSHL R2
D273 CF 06 FB 0006D CALLS #6, DBG$BUILD_PRIMARY_SUBNODE
04 00072 RET
```


DBGPARSER
V04-000

D 6
16-Sep-1984 02:10:13
14-Sep-1984 12:17:30

VAX-11 Bliss-32 V4.0-742
[DEBUG.SRC]DBGPARSER.B32;1

Page 316
(38)

; Routine Size: 115 bytes, Routine Base: DBG\$CODE + 2087

```
9829 9924 1 ROUTINE GET_FIELDREF (TOKEN): NOVALUE =
9830 9925 1
9831 9926 1 FUNCTION
9832 9927 1 This routine picks up the position, size, and (optionally) the
9833 9928 1 extension in a field reference (i.e., XX<pos,size,ext>.
9834 9929 1 DBG$EXPRESSION_PARSER is called to parse and evaluate each of
9835 9930 1 these values. The values are stored as integers in the
9836 9931 1 Operator Lexical Token entry.
9837 9932 1
9838 9933 1 This routine assumes that the opening angle bracket has already
9839 9934 1 been found and that the parse pointer points to the start of
9840 9935 1 the first expression in the field reference. When this routine
9841 9936 1 returns, the parse pointer is left at the first character after
9842 9937 1 the closing angle bracket.
9843 9938 1
9844 9939 1 INPUTS
9845 9940 1 TOKEN - a pointer to the Operator Lexical Token entry for
9846 9941 1 the "<" operator.
9847 9942 1
9848 9943 1 OUTPUTS
9849 9944 1 The Lexical Token pointed to by TOKEN is modified to include the
9850 9945 1 offset, size, and sign extension information.
9851 9946 1
9852 9947 1
9853 9948 1 BEGIN
9854 9949 1
9855 9950 1 MAP
9856 9951 1 TOKEN: REF TOKEN$ENTRY; | Pointer to the Lexical Token Entry
9857 9952 1 | for the field reference operator
9858 9953 1
9859 9954 1 LOCAL
9860 9955 1 DECLTYPE: REF DBG$VALDESC, | Pointer to Value Descriptor
9861 9956 1 | for one of the values
9862 9957 1 | inside the angle brackets
9863 9958 1 SAVED_RADIX, | Temporarily saved expression radix
9864 9959 1 VALUE; | Value of position, size, or sign ext
9865 9960 1 | field
9866 9961 1 VALPTR: REF DBG$VALDESC; | Pointer to position, size, or
9867 9962 1 | extension value descriptor.
9868 9963 1
9869 9964 1
9870 9965 1
9871 9966 1 | Loop through the expressions in this field reference. Each of these
9872 9967 1 | is parsed and evaluated via a call to DBG$EXPRESSION_PARSER.
9873 9968 1 | This returns a descriptor, and the type converter is then called
9874 9969 1 | to convert the descriptor into an integer value.
9875 9970 1 | The integer value is checked for being in an appropriate range and
9876 9971 1 | then stored in the appropriate own variable.
9877 9972 1
9878 9973 1 INCR I FROM 1 TO 3 DO
9879 9974 1 BEGIN
9880 9975 1
9881 9976 1 | Call the expression parser to pick up the next expression in the
9882 9977 1 | field reference. Note that we set the radix to decimal over this
9883 9978 1 | call and then restore it. Also note that the Expression Parser sets
9884 9979 1 | TERMINATOR_CODE and TERMINATOR_LENGTH as a side-effect.
9885 9980 1
```

```

9886
9887
9888
9889
9890
9891
9892
9893
9894
9895
9896
9897
9898
9899
9900
9901
9902
9903
9904
9905
9906
9907
9908
9909
9910
9911
9912
9913
9914
9915
9916
9917
9918
9919
9920
9921
9922
9923
9924
9925
9926
9927
9928
9929
9930
9931
9932
9933
9934
9935
9936
9937
9938
9939
9940
9941
9942

```

```

!
! SAVED RADIX = .EXPRESSION RADIX;
! EXPRESSION RADIX = DBG$K DECIMAL;
! VALPTR = DBG$EXPRESSION_PARSER(FALSE, BIT_SELECT_TERM_TBL);
! EXPRESSION_RADIX = .SAVED_RADIX;

! Check the terminator code. If there was no terminator (i.e., the
! input line just ended), signal an error. Otherwise we got a comma
! or closing angle bracket and we increment CHARPTR to get past it.
! IF .TERMINATOR_CODE EQL TOKEN$K_TERM_NONE THEN SIGNAL(DBG$MISCLOSUB);
! CHARPTR = .CHARPTR + .TERMINATOR_LENGTH;

! Convert the value descriptor to an integer.
! VALUE = CONVERT_TO_INTEGER (.VALPTR);

! If we are looking at the "position" field, it can be any value that
! fits in the PRIM_OFFSET field of the Primary Descriptor. We do not
! allow any terminator other than comma in this case.
! IF .I EQL 1
! THEN
! BEGIN
! IF .VALUE LSS 'X'0000' OR .VALUE GTR 'X'7FFF'
! THEN
! SIGNAL (DBG$_ILLPOSFLD, 1, .VALUE);
!
! TOKEN [TOKEN$W_BIT_OFFSET] = .VALUE;
! IF .TERMINATOR_CODE EQL TOKEN$K_TERM_GTRTHAN
! THEN
! SIGNAL(DBG$_INVFLDREF);
!
! END;

! If we are looking at the size field, the value must be in the
! range 0 to 32 inclusive. Here we exit the loop normally on a
! closing angle bracket, defaulting the sign extension to zero.
! IF .I EQL 2
! THEN
! BEGIN
! IF .VALUE LSS 0 OR .VALUE GTR 32
! THEN
! SIGNAL (DBG$_ILLSIZFLD, 1, .VALUE);
!
! TOKEN [TOKEN$W_BIT_LENGTH] = .VALUE;
! IF .TERMINATOR_CODE EQL TOKEN$K_TERM_GTRTHAN THEN EXITLOOP;
! END;

! If we are looking at the sign extension field then the value
! must be either 0 or 1. We also insist that the terminator must

```

9943 10038
9944 10039
9945 10040
9946 10041
9947 10042
9948 10043
9949 10044
9950 10045
9951 10046
9952 10047
9953 10048
9954 10049
9955 10050
9956 10051
9957 10052
9958 10053
9959 10054
9960 10055
9961 10056
9962 10057
9963 10058
9964 10059
9965 10060
9966 10061

```
! be the closing angle bracket in this case.
IF .I EQL 3
THEN
  BEGIN
    IF .VALUE NEQ 0 AND .VALUE NEQ 1
    THEN
      SIGNAL (DBG$_ILLSIGEXT, 1, .VALUE);

      TOKEN [TOKEN$_V_SGNEXT] = .VALUE;
      IF .TERMINATOR_CODE NEQ TOKEN$_K_TERM_GTRTHAN
      THEN
        SIGNAL (DBG$_INVFLDREF);

      EXITLOOP;
    END;

  END;

! All done. Return to the caller.
RETURN;
END;
```

00FC 0000 GET_FIELDREF:							
	57	00000000G	00 9E 00002	WORD	Save R2,R3,R4,R5,R6,R7	9924	
	56	00000000'	EF 9E 00009	MOVAB	LIB\$SIGNAL, R7		
	53		01 D0 00010	MOVAB	TERMINATOR_CODE, R6		
	54	DB	A6 D0 00013	1\$:	MOVL	#1, I	9973
DB	A6		0A D0 00017	MOVL	EXPRESSION_RADIX, SAVED_RADIX	9982	
		00000000'	EF 9F 0001B	MOVL	#10, EXPRESSION_RADIX	9983	
			7E D4 00021	PUSHAB	BIT_SELECT_TERM_TBL	9984	
			02 FB 00023	CLRL	-(SP)		
D7D6	CF		50 D0 00028	CALLS	#2, DBG\$EXPRESSION_PARSER		
	55		54 D0 0002B	MOVL	R0, VALPTR		
DB	A6		66 D5 0002F	MOVL	SAVED_RADIX, EXPRESSION_RADIX	9985	
			09 12 00031	TSTL	TERMINATOR_CODE	9992	
		00028E90	8F DD 00033	BNEQ	2\$		
	67		01 FB 00039	PUSHL	#167568		
FBCC	C6	04	A6 C0 0003C	2\$:	CALLS	#1, LIB\$SIGNAL	9993
			55 DD 00042	ADDL2	TERMINATOR_LENGTH, CHARPTR		
F65A	CF		01 FB 00044	PUSHL	VALPTR	9998	
	52		50 D0 00049	CALLS	#1, CONVERT_TO_INTEGER		
	01		53 D1 0004C	MOVL	R0, VALUE		
			30 12 0004F	CMPL	I, #1	1000	
			52 D5 00051	BNEQ	5\$		
			09 19 00053	TSTL	VALUE	1000	
00007FFF	8F		52 D1 00055	BLSS	3\$		
			0D 15 0005C	CMPL	VALUE, #32767		
			52 DD 0005E	BLEQ	4\$		
			01 DD 00060	3\$:	PUSHL	VALUE	1001
				PUSHL	#1		

			00028EC8	8F	DD	00062	PUSHL	#167624	
	67			03	FB	00068	CALLS	#3, LIB\$SIGNAL	
	50		04	AC	D0	0006B	4\$:	MOVL	TOKEN, R0
08	0A			52	B0	0006F		MOVW	VALUE, 8(R0)
	0A			66	D1	00073		CMPL	TERMINATOR_CODE, #10
				09	12	00076		BNEQ	5\$
			00028F00	8F	DD	00078		PUSHL	#167680
	67			01	FB	0007E		CALLS	#1, LIB\$SIGNAL
	02			53	D1	00081	5\$:	CMPL	I, #2
				23	12	00084		BNEQ	8\$
				52	D5	00086		TSTL	VALUE
				05	19	00088		BLSS	6\$
	20			52	D1	0008A		CMPL	VALUE, #32
				0D	15	0008D		BLEQ	7\$
				52	DD	0008F	6\$:	PUSHL	VALUE
				01	DD	00091		PUSHL	#1
			00028ED0	8F	DD	00093		PUSHL	#167632
	67			03	FB	00099		CALLS	#3, LIB\$SIGNAL
	50		04	AC	D0	0009C	7\$:	MOVL	TOKEN, R0
0A	0A			52	B0	000A0		MOVW	VALUE, 10(R0)
	0A			66	D1	000A4		CMPL	TERMINATOR_CODE, #10
				36	13	000A7		BEQL	11\$
	03			53	D1	000A9	8\$:	CMPL	I, #3
				2B	12	000AC		BNEQ	10\$
				52	D5	000AE		TSTL	VALUE
				12	13	000B0		BEQL	9\$
	01			52	D1	000B2		CMPL	VALUE, #1
				0D	13	000B5		BEQL	9\$
				52	DD	000B7		PUSHL	VALUE
				01	DD	000B9		PUSHL	#1
			00028140	8F	DD	000BB		PUSHL	#164160
	67			03	FB	000C1		CALLS	#3, LIB\$SIGNAL
04	0A			52	F0	000C4	9\$:	INSV	VALUE, #10, #1, @TOKEN
	0A			66	D1	000CA		CMPL	TERMINATOR_CODE, #10
				10	13	000CD		BEQL	11\$
			00028F00	8F	DD	000CF		PUSHL	#167680
	67			01	FB	000D5		CALLS	#1, LIB\$SIGNAL
					04	000D8		RET	
					F1	000D9	10\$:	ACBL	#3, #1, I, 1\$
				04	000DF	11\$:		RET	

; Routine Size: 224 bytes, Routine Base: DBG\$CODE + 2DFA

```
9968 10062 1
9969 10063 1
9970 10064 1
9971 10065 1
9972 10066 1
9973 10067 1
9974 10068 1
9975 10069 1
9976 10070 1
9977 10071 1
9978 10072 1
9979 10073 1
9980 10074 1
9981 10075 1
9982 10076 1
9983 10077 1
9984 10078 1
9985 10079 1
9986 10080 1
9987 10081 1
9988 10082 1
9989 10083 1
9990 10084 1
9991 10085 1
9992 10086 1
9993 10087 1
9994 10088 1
9995 10089 1
9996 10090 1
9997 10091 1
9998 10092 1
9999 10093 2
10000 10094 2
10001 10095 2
10002 10096 2
10003 10097 2
10004 10098 2
10005 10099 2
10006 10100 2
10007 10101 2
10008 10102 2
10009 10103 2
10010 10104 2
10011 10105 2
10012 10106 2
10013 10107 2
10014 10108 2
10015 10109 2
10016 10110 2
10017 10111 2
10018 10112 2
10019 10113 2
10020 10114 2
10021 10115 2
10022 10116 2
10023 10117 2
10024 10118 2
```

ROUTINE GET_RECORD_COMPONENT(PRIMPTR, COMPNAME): NOVALUE =

FUNCTION

This routine is called during the parsing of Primary Symbols to do record component selection. It accepts as input a Primary Descriptor for a record and the name of a record component to be selected from that record. It then checks that the Primary Descriptor is indeed for a record (otherwise component selection is not allowed and an error is signalled). It then looks up the component name in the RST and gets the SYMID for the specified component of the specified record. If no such component exists for this record, an error is signalled. Finally, this component SYMID is converted to a record component index which is stored in the Record Sub-Node in the Primary Descriptor and another Sub-Node is appended for the record component itself. The output of the routine is thus the side-effect of modifying the input Primary Descriptor.

INPUTS

PRIMPTR - A pointer to the Primary Descriptor for the record on which component selection is to be done.

COMPNAME - A pointer to the name of the record component to be selected. The name must be in Counted ASCII format.

OUTPUTS

The PRIMPTR Primary Descriptor is modified by filling in the record component index for the selected component and by appending another Primary Descriptor Sub-Node for the component. The PRIMPTR pointer itself is not modified, however.

BEGIN

MAP

PRIMPTR: REF DBG\$PRIMARY; ! Pointer to Primary Descriptor

LOCAL

COMP_LIST: REF VECTOR[]; ! List of potential
component symids
COMP_LIST_SIZE, ! Length of COMP_LIST
COMPSYMID: REF RST\$ENTRY, ! SYMID for current record component
EXACT_MATCH, ! Flag saying we've found the record component
FCODE, ! The FCODE of the record component
NODEPTR: REF DBG\$PRIM_NODE, ! Pointer to Record Sub-Node in the
Primary Descriptor
STATUS, ! Status returned by GET_RECORD_VARIANT
SYMID: REF RST\$ENTRY, ! The SYMID of the record component
TYP_COMPLST: REF VECTOR[.LONG], ! Pointer to list of record component
SYMIDs in record Type RST Entry
TYPEID: REF RST\$ENTRY, ! The Type ID of the record record type
or of the record component
VARPTR: REF RST\$VAR_ENTRY, ! Pointer to current RST Variant Entry
VARSETPTR: REF RST\$ENTRY; ! Pointer to Variant Set RST Entry

DBG\$GL_CURRENT_PRIMARY = .PRIMPTR;

```
:10025      10119  2
:10026      10120  2
:10027      10121  2
:10028      10122  2
:10029      10123  2
:10030      10124  2
:10031      10125  2
:10032      10126  2
:10033      10127  2
:10034      10128  2
:10035      10129  2
:10036      10130  2
:10037      10131  2
:10038      10132  2
:10039      10133  2
:10040      10134  2
:10041      10135  2
:10042      10136  2
:10043      10137  2
:10044      10138  2
:10045      10139  2
:10046      10140  2
:10047      10141  2
:10048      10142  2
:10049      10143  2
:10050      10144  2
:10051      10145  2
:10052      10146  2
:10053      10147  2
:10054      10148  2
:10055      10149  2
:10056      10150  2
:10057      10151  2
:10058      10152  2
:10059      10153  2
:10060      10154  2
:10061      10155  2
:10062      10156  2
:10063      10157  2
:10064      10158  2
:10065      10159  2
:10066      10160  2
:10067      10161  2
:10068      10162  2
:10069      10163  2
:10070      10164  2
:10071      10165  2
:10072      10166  2
:10073      10167  2
:10074      10168  2
:10075      10169  2
:10076      10170  2
:10077      10171  2
:10078      10172  2
:10079      10173  2
:10080      10174  2
:10081      10175  2
```

```
! Check that the Primary Descriptor is for a record--otherwise record
! component selection is not allowed.
```

```
NODEPTR = .PRIMPTR[DBG$L_PRIM_BLINK];
IF .ENFORCE_RECORD
THEN
  BEGIN
```

```
! For ADA, the pointer dereference is implicit. That is, if
! PTR is a pointer then PTR.C is equivalent to PASCAL's
! PTR^.C
```

```
IF .DBG$GB_LANGUAGE EQL DBG$K_ADA
THEN
  BEGIN
```

```
! First check for the special case ".ALL", which
! means pointer dereference.
```

```
IF CH$EQL(4, .COMPNAME,
          4, UPLIT BYTE (3, 'A', 'L', 'L'))
```

```
THEN
  BEGIN
    GET_DEREFERENCE(.PRIMPTR);
    RETURN;
  END;
```

```
! Now do implicit dereference of the pointer.
```

```
WHILE .PRIMPTR[DBG$B_DHDR_FCODE] EQL RST$K_TYPE_TPTR DO
  GET_DEREFERENCE(.PRIMPTR);
NODEPTR = .PRIMPTR[DBG$L_PRIM_BLINK];
END;
```

```
IF .PRIMPTR[DBG$B_DHDR_FCODE] NEQ RST$K_TYPE_RECORD
THEN
  SIGNAL(DBG$NOTRECORD, 1, .COMPNAME);
```

```
IF .NODEPTR[DBG$B_PNODE_FCODE] NEQ RST$K_TYPE_RECORD
THEN
  $DBG_ERROR('DBGPARSER\GET_RECORD_COMPONENT 10');
END
```

```
! For languages which allow non-records to be component-selected
! (i.e., language C), we do not do the above check. In fact,
! we explicitly change the FCODE to say "record" so that
! the Primary is processed as a record by routines in DBGVALUES
! and in DBGPRINT.
```

```
ELSE
  NODEPTR[DBG$B_PNODE_FCODE] = RST$K_TYPE_RECORD;
```

```
! Search the RST Hash Table for all record components of this name.
! If we find one which belongs to the given record, then light the
```

```
10082 10176 2
10083 10177
10084 10178
10085 10179
10086 10180
10087 10181
10088 10182
10089 10183
10090 10184
10091 10185
10092 10186
10093 10187
10094 10188
10095 10189
10096 10190
10097 10191
10098 10192
10099 10193
10100 10194
10101 10195
10102 10196
10103 10197
10104 10198
10105 10199
10106 10200
10107 10201
10108 10202
10109 10203
10110 10204
10111 10205
10112 10206
10113 10207
10114 10208
10115 10209
10116 10210
10117 10211
10118 10212
10119 10213
10120 10214
10121 10215
10122 10216
10123 10217
10124 10218
10125 10219
10126 10220
10127 10221
10128 10222
10129 10223
10130 10224
10131 10225
10132 10226
10133 10227
10134 10228
10135 10229
10136 10230
10137 10231
10138 10232 2

! EXACT_MATCH flag and exit the loop. If we find one but it belongs
! to the wrong record, we save its SYMID anyway - some languages
! allow this. If we find more than one but they all belong to the
! wrong record, then light DUPLICATE_FLAG - this will be an error.
TYPEID = .NODEPTR[DBG$PNODE_TYPEID];
SYMID = 0;
EXACT_MATCH = FALSE;
COMP_LIST_SIZE = 10;
COMP_LIST = DBG$GET_TEMPMEM(.COMP_LIST_SIZE);
COMP_LIST[0] = 0;
DBG$HASH_FIND_SETUP(.COMPNAME);
WHILE TRUE DO
  BEGIN
    SYMID = DBG$HASH_FIND(.COMPNAME);
    IF .SYMID EQL 0 THEN EXITLOOP;
    IF .SYMID[RST$B_KIND] EQL RST$K_TYPCOMP
    THEN
      BEGIN
        IF .SYMID[RST$L_UPSCOPEPTR] EQL .TYPEID
        THEN
          BEGIN
            ! This is the case where we find a component with the right
            ! name belonging to the right record.
            EXACT_MATCH = TRUE;
            EXITLOOP;
          END
        ELSE
          BEGIN
            COMP_LIST[0] = .COMP_LIST[0] + 1;
            ! If we overflow the component list then expand it.
            IF .COMP_LIST[0] GEQ .COMP_LIST_SIZE
            THEN
              BEGIN
                LOCAL
                SAVE COMP_LIST;
                SAVE_COMP_LIST = .COMP_LIST;
                COMP_LIST = DBG$GET_TEMPMEM(.COMP_LIST_SIZE+10);
                CHSMOVE(4*.COMP_LIST_SIZE, .SAVE_COMP_LIST, .COMP_LIST);
                COMP_LIST_SIZE = .COMP_LIST_SIZE + 10;
              END;
              COMP_LIST[.COMP_LIST[0]] = .SYMID;
            END;
          END;
        END;
      END;
    END;
  END;

! Signal errors for the cases where a record component was not found,
! or where the reference is ambiguous.
IF NOT .EXACT_MATCH
```



```
:10139
:10140
:10141
:10142
:10143
:10144
:10145
:10146
:10147
:10148
:10149
:10150
:10151
:10152
:10153
:10154
:10155
:10156
:10157
:10158
:10159
:10160
:10161
:10162
:10163
:10164
:10165
:10166
:10167
:10168
:10169
:10170
:10171
:10172
:10173
:10174
:10175
:10176
:10177
:10178
:10179
:10180
:10181
:10182
:10183
:10184
:10185
:10186
:10187
:10188
:10189
:10190
:10191
:10192
:10193
:10194
:10195
```

```
10233
10234
10235
10236
10237
10238
10239
10240
10241
10242
10243
10244
10245
10246
10247
10248
10249
10250
10251
10252
10253
10254
10255
10256
10257
10258
10259
10260
10261
10262
10263
10264
10265
10266
10267
10268
10269
10270
10271
10272
10273
10274
10275
10276
10277
10278
10279
10280
10281
10282
10283
10284
10285
10286
10287
10288
10289
```

```
THEN
  IF .ENFORCE_RECORD AND NOT .INCOMPLETE_QUAL
  THEN
    ! We did not find a component in the right record, and this
    ! language is strict about membership checking. So signal
    ! an error.
    SIGNAL(DBG$_NOFIELD, 1, .COMPNAME)
  ELSE
    IF .COMP_LIST[0] EQL 0
    THEN
      ! We did not find any components at all. Signal an error.
      SIGNAL(DBG$_NOFIELD, 1, .COMPNAME)
    ELSE
      ! Call a routine which attempts to resolve amiguities.
      ! If it fails, it will return false and we signal an
      ! error saying that the record reference was ambiguous.
      IF NOT RESOLVE_COMPONENT(.TYPEID, .COMP_LIST, SYMID,
                              .PRIMPTR, .COMPNAME)
      THEN
        SIGNAL(DBG$_AMBFIELD, 1, .COMPNAME);

      ! Set the EVAL bit in the Record Sub-Node to indicate that a record com-
      ! ponent has actually been selected.
      NODEPTR = .PRIMPTR[DBG$$_PRIM.BLINK];
      NODEPTR[DBG$$_PNODE_EVAL] = TRUE;

      ! We have found a Type Component SYMID for a component of the current
      ! record data type. Now convert that SYMID to a record component index
      ! into the component vector for the record type. We do this by searching
      ! the component list in the record type's Type RST Entry.
      TYP_COMPLST = TYPEID[RST$_TYP_COMPLST];
      INCR I FROM 0 TO .TYPEID[RST$_TYP_COMPCNT] - 1 DO
        BEGIN
          COMPSYMID = .TYP_COMPLST[I];

          ! If this component is the one we seek, set its index into the Record
          ! Sub-Node and leave the loop.
          IF .SYMID EQL .COMPSYMID
          THEN
            BEGIN
              NODEPTR[DBG$$_PNREC_INDEX] = .I + 1;
```

```
10196      EXITLOOP;
10197      END;
10198
10199      ! If this record component is a Variant Set, see if the desired
10200      ! component is part of one of the variants in this Variant Set.
10201      IF .COMPSYMLD[RST$B_KIND] EQL RST$K_VARIANT
10202      THEN
10203      BEGIN
10204      VARSTK_INDEX = 0;
10205      STATUS = GET_RECORD_VARIANT(.COMPSYMLD, .SYMID);
10206      IF .STATUS
10207      THEN
10208      BEGIN
10209
10210      ! We found the record component in the current Variant Set.
10211      ! Set the index of the Variant Set in the Record Sub-Node.
10212      ! Then build all necessary Primary Descriptor Variant Sub-
10213      ! Nodes, one for each level of variant nesting.
10214      NODEPTR[DBG$W_PNREC_INDEX] = .I + 1;
10215      INCR J FROM 0 TO .VARSTK_INDEX - 1 DO
10216      BEGIN
10217      DBG$BUILD_PRIMARY_SUBNODE(.PRIMPTR, RST$K_VARIANT,
10218      0, RST$K_TYPE_VARIANT, 0, 0);
10219      VARSETPTR = .VARSTACK1[J];
10220      VARPTR = .VARSTACK2[J];
10221      NODEPTR = .PRIMPTR[DBG$W_PNREC_INDEX];
10222      NODEPTR[DBG$W_PNREC_INDEX] = TRUE;
10223      NODEPTR[DBG$W_PNVAR_TAGID] = .VARSETPTR[RST$L_VARTAGPTR];
10224      NODEPTR[DBG$W_PNVAR_INDEX] = .VARSTACK3[J];
10225      NODEPTR[DBG$W_PNVAR_NCOMPS] = .VARPTR[RST$L_VAR_COMPCNT];
10226      NODEPTR[DBG$W_PNVAR_COMPLST] = .VARPTR[RST$L_VAR_COMPLST];
10227      NODEPTR[DBG$W_PNVAR_DSTPTR] = .VARPTR[RST$L_VAR_DSTPTR];
10228      END;
10229
10230      ! The Variant Sub-Nodes have successfully been constructed.
10231      ! Now exit the search of the record component list so that
10232      ! we can build the Sub-Node for the component actually found.
10233      EXITLOOP;
10234      END;
10235      END;
10236      ! End of variant code
10237      END;
10238      ! End of loop over record components
10239
10240      ! Finally append another Primary Descriptor Sub-Node for the selected
10241      ! record component. Then return.
10242      DBG$STA_SYMTYPE(.SYMID, FCODE, TYPEID);
10243      DBG$BUILD_PRIMARY_SUBNODE(.PRIMPTR,
10244      RST$K_TYPCOMP, .SYMID, .FCODE, .TYPEID, 0);
```

```

.PSECT DBG$PLIT,NOWRT, SHR, PIC.0

```

```

03 03372 P.AYJ: .BYTE 3
41 03373 .ASCII \A\
4C 03374 .ASCII \L\
4C 03375 .ASCII \L\
21 03376 P.AYK: .ASCII \!DBGPARSER\<92>\GET_RECORD_COMPONENT 10\
52 03385
54 03394

```

.PSECT DBG\$CODE,NOWRT, SHR, PIC.0

OFFC 00000 GET_RECORD_COMPONENT:

Offset	Hex	Symbol	Disassembly	Comment	Address
00000000G	5E	10	C2 00002	WORD	1006
	5A	04	AC D0 00005	SUBL2	
	00		5A D0 00009	MOVL	1011
	57	18	AA D0 00010	MOVL	1012
	63	00000000'	EF E9 00014	BLBC	1012
	09	00000000G	00 91 0001B	CMPB	1013
			25 12 00022	BNEQ	
00000000'	EF	08	BC D1 00024	CMPL	1013
			08 12 0002C	BNEQ	
			5A DD 0002E	PUSHL	1014
FE78	CF		01 FB 00030	CALLS	
			04 00035	RET	1014
	06	06	AA 91 00036	CMPB	1014
			09 12 0003A	BNEQ	
			5A DD 0003C	PUSHL	1015
FE6A	CF		01 FB 0003E	CALLS	
			F1 11 00043	BRB	
	57	18	AA D0 00045	MOVL	1015
	07	06	AA 91 00049	CMPB	1015
			12 13 0004D	BEQL	
		08	AC DD 0004F	PUSHL	1015
			01 DD 00052	PUSHL	
		00028E98	8F DD 00054	PUSHL	
00000000G	00		03 FB 0005A	CALLS	
	07	09	A7 91 00061	CMPB	1015
			1B 13 00065	BEQL	
		00000000'	EF 9F 00067	PUSHAB	1016
			01 DD 0006D	PUSHL	
		00028362	8F DD 0006F	PUSHL	
00000000G	00		03 FB 00075	CALLS	
			04 11 0007C	BRB	1012
	09	A7	07 90 0007E	MOVB	1017
	08	0C	A7 D0 00082	MOVL	1018
			6E 7C 00087	CLRQ	1018
	56		0A D0 00089	MOVL	1018

00000000G	00		56	DD	0008C	PUSHL	COMP_LIST_SIZE	1018
	59		01	FB	0008E	CALLS	#1, DBG\$GET_TEMPHEM	
			50	D0	00095	MOVL	R0, COMP_LIST	
			69	D4	00098	CLRL	(COMP_LIST)	1018
	5B	08	AC	D0	0009A	MOVL	COMPNAME, R11	1018
			5B	DD	0009E	PUSHL	R11	
00000000G	00		01	FB	000A0	CALLS	#1, DBG\$HASH_FIND_SETUP	
			5B	DD	000A7	PUSHL	R11	1019
00000000G	00		01	FB	000A9	CALLS	#1, DBG\$HASH_FIND	
04	AE		50	D0	000B0	MOVL	R0, SYMID	
	58	04	AE	D0	000B4	MOVL	SYMID, R8	1019
			3D	13	000B8	BEQL	10\$	
	0A	14	A8	91	000BA	CMPB	20(R8), #10	1019
			E7	12	000BE	BNEQ	7\$	
08	AE	10	A8	D1	000C0	CMPL	16(R8), TYPEID	1019
			05	12	000C5	BNEQ	8\$	
	6E		01	D0	000C7	MOVL	#1, EXACT_MATCH	1020
			2B	11	000CA	BRB	10\$	1019
			69	D6	000CC	INCL	(COMP_LIST)	1020
	56		69	D1	000CE	CMPL	(COMP_LIST), COMP_LIST_SIZE	1021
			1B	19	000D1	BLSS	9\$	
	52		59	D0	000D3	MOVL	COMP_LIST, SAVE_COMP_LIST	1021
		0A	A6	9F	000D6	PUSHAB	10(COMP_LIST_SIZE)	1021
00000000G	00		01	FB	000D9	CALLS	#1, DBG\$GET_TEMPHEM	
	59		50	D0	000E0	MOVL	R0, COMP_LIST	
50	56		02	78	000E3	ASHL	#2, COMP_LIST_SIZE, R0	1022
69	62		50	28	000E7	MOVC3	R0, (SAVE_COMP_LIST), (COMP_LIST)	
	56		0A	C0	000EB	ADDL2	#10, COMP_LIST_SIZE	1022
	50		69	D0	000EE	MOVL	(COMP_LIST), R0	1022
	6940		58	D0	000F1	MOVL	R8, (COMP_LIST)[R0]	
			B0	11	000F5	BRB	7\$	1018
	42		6E	E8	000F7	BLBS	EXACT_MATCH, 15\$	1023
	07	00000000'	EF	E9	000FA	BLBC	ENFORCE_RECORD, 11\$	1023
	04	00000000'	EF	E9	00101	BLBC	INCOMPLETE_QUAL, 12\$	
			69	D5	00108	TSTL	(COMP_LIST)	1024
			0C	12	0010A	BNEQ	13\$	
			5B	DD	0010C	PUSHL	R11	1025
			01	DD	0010E	PUSHL	#1	
	00028C80		8F	DD	00110	PUSHL	#167040	
			1D	11	00116	BRB	14\$	
	7E		5A	7D	00118	MOVQ	R10, -(SP)	1026
		0C	AE	9F	0011B	PUSHAB	SYMID	1025
			59	DD	0011E	PUSHL	COMP_LIST	
		18	AE	DD	00120	PUSHL	TYPEID	
0000V	CF		05	FB	00123	CALLS	#5, RESOLVE_COMPONENT	
	11		50	E8	00128	BLBS	R0, 15\$	
			5B	DD	0012B	PUSHL	R11	1026
			01	DD	0012D	PUSHL	#1	
		00028F58	8F	DD	0012F	PUSHL	#167768	
00000000G	00		03	FB	00135	CALLS	#3, LIB\$SIGNAL	
	57	18	AA	D0	0013C	MOVL	24(R10), NODEPTR	1026
	0A		01	88	00140	BISB2	#1, 10(NODEPTR)	1026
	58	08	AE	D0	00144	MOVL	TYPEID, R8	1027
	52	2C	A8	9E	00148	MOVAB	44(R8), TYPcomplst	
	54		01	CE	0014C	MNEGL	#1, I	1028
			D086	31	0014F	BRW	20\$	
	56		6244	D0	00152	MOVL	(TYPcomplst)[I], COMPSYMID	1028

18	A7	56	04	AE	D1	00156	CMPL	SYMID, COMPSYMID	1028	
		07		12	0015A	BNEQ	17\$			
		54		01	A1	0015C	ADDW3	#1, 1, 24(NODEPTR)	1028	
		7F		11	00161	BRB	22\$		1028	
		0B	14	A6	91	00163	17\$: CMPB	20(COMPSYMID), #11	1029	
		6F		12	00167	BNEQ	20\$			
		00000000'		EF	D4	00169	CLRL	VARSTK_INDEX	1030	
		04		AE	DD	0016F	PUSHL	SYMID	1030	
		56		DD	00172	PUSHL	COMPSYMID			
	0000V	CF		02	FB	00174	CALLS	#2, GET_RECORD_VARIANT		
	6E	50		DD	00179	MOVL	R0, STATUS			
	59	6E		E9	0017C	BLBC	STATUS, 20\$		1030	
18	A7	54		01	A1	0017F	ADDW3	#1, 1, 24(NODEPTR)	1031	
		5B	00000000'	EF	DD	00184	MOVL	VARSTK_INDEX, R11	1031	
		53		01	CE	0018B	MNEGL	#1, J		
				42	11	0018E	BRB	19\$		
				7E	7C	00190	18\$: CLRL	-(SP)	1031	
				13	DD	00192	PUSHL	#19		
		7E		0B	7D	00194	MOVQ	#11, -(SP)		
				5A	DD	00197	PUSHL	R10		
	CFF4	CF		06	FB	00199	CALLS	#6, DBG\$BUILD_PRIMARY_SUBNODE		
		59	00000000'	EF	43	DD	0019E	MOVL	VARSTACK1[J], VARSETPTR	1031
		55	00000000'	EF	43	DD	001A6	MOVL	VARSTACK2[J], VARPTR	1031
		57	18	AA	DD	001AE	MOVL	24(R10), NODEPTR	1031	
	0A	A7		01	88	001B2	BISB2	#1, 10(NODEPTR)	1032	
	1C	A7	10	A9	DD	001B6	MOVL	16(VARSETPTR), 28(NODEPTR)	1032	
	18	A7	00000000'	EF	43	F7	001BB	CVTLW	VARSTACK3[J], 24(NODEPTR)	1032
	1A	A7	04	A5	B0	001C4	MOVW	4(VARPTR), 26(NODEPTR)	1032	
	20	A7	0B	A5	9E	001C9	MOVAB	8(R5), 32(NODEPTR)	1032	
	24	A7		65	DD	001CE	MOVL	(VARPTR), 36(NODEPTR)	1032	
BA		53		5B	F2	001D2	19\$: AOBLSS	R11, J, 18\$	1031	
				0A	11	001D6	BRB	22\$	1030	
02		54	2B	A8	F2	001D8	20\$: AOBLSS	40(R8), 1, 21\$	1027	
				03	11	001DD	BRB	22\$		
				FF	70	31	001DF	21\$: BRW	16\$	
				08	AE	9F	001E2	22\$: PUSHAB	TYPEID	1034
				10	AE	9F	001E5	PUSHAB	FCODE	
				0C	AE	DD	001E8	PUSHL	SYMID	
	00000000G	00		03	FB	001EB	CALLS	#3, DBG\$STA_SYMTYPE		
				7E	D4	001F2	CLRL	-(SP)	1034	
				0C	AE	DD	001F4	PUSHL	TYPEID	1034
				14	AE	DD	001F7	PUSHL	FCODE	
				10	AE	DD	001FA	PUSHL	SYMID	
				0A	DD	001FD	PUSHL	#10	1034	
				5A	DD	001FF	PUSHL	R10		
	CF8C	CF		06	FB	00201	CALLS	#6, DBG\$BUILD_PRIMARY_SUBNODE		
				04	00206	RET			1034	

; Routine Size: 519 bytes, Routine Base: DBG\$CODE + 2EDA

```
:10257 10350 1
:10258 10351 1
:10259 10352 1
:10260 10353 1
:10261 10354 1
:10262 10355 1
:10263 10356 1
:10264 10357 1
:10265 10358 1
:10266 10359 1
:10267 10360 1
:10268 10361 1
:10269 10362 1
:10270 10363 1
:10271 10364 1
:10272 10365 1
:10273 10366 1
:10274 10367 1
:10275 10368 1
:10276 10369 1
:10277 10370 1
:10278 10371 1
:10279 10372 1
:10280 10373 1
:10281 10374 1
:10282 10375 1
:10283 10376 1
:10284 10377 1
:10285 10378 1
:10286 10379 1
:10287 10380 2
:10288 10381 2
:10289 10382 2
:10290 10383 2
:10291 10384 2
:10292 10385 2
:10293 10386 2
:10294 10387 2
:10295 10388 2
:10296 10389 2
:10297 10390 2
:10298 10391 2
:10299 10392 2
:10300 10393 2
:10301 10394 2
:10302 10395 2
:10303 10396 2
:10304 10397 2
:10305 10398 2
:10306 10399 2
:10307 10400 2
:10308 10401 2
:10309 10402 2
:10310 10403 2
:10311 10404 2
:10312 10405 2
:10313 10406 2
```

ROUTINE GET_RECORD_VARIANT(VARSETPTR, SYMID) =

FUNCTION

This routine looks for a record component with a known SYMID among all the variants in a specified Variant Set. It returns TRUE if the component is found in that Variant Set, and as a side-effect, it also builds a "Variant Stack" which specifies which sequence of variants and Variant Sets contain the component. This routine calls itself recursively to search variants within variants; the "Variant Stack" records the path taken through the tree of variants to reach the desired component. This stack is then used by GET_RECORD_COMPONENT to build the Primary Descriptor Variant Sub-Nodes needed to describe that path.

INPUTS

VARSETPTR - Pointer to the Variant Set to be searched for the SYMID record component.

SYMID - The SYMID (RST Entry address) of the record component to search for among the variants of the current Variant Set. (This SYMID has been found by looking up the record component by its name.)

OUTPUTS

If the SYMID record component is found in any of the VARSETPTR variants, this routine returns TRUE as its value. If the SYMID component is not found, it returns FALSE. If TRUE is returned, the Variant Stack (in OWN storage) contains the list of Variant Sets and variants which contain the SYMID component.

BEGIN

MAP

VARSETPTR: REF RST\$ENTRY; ! Pointer to Variant Set RST Entry

LOCAL

COMPLST: REF VECTOR[.LONG], ! Pointer to vector of component RST
pointers in RST Variant Entry
COMPTR: REF RST\$ENTRY, ! Pointer to current variant component's
RST entry
VARPTR: REF RST\$VAR ENTRY, ! Pointer to current RST Variant Entry
VARSETIBL: REF VECTOR[.LONG], ! Pointer to vector of variant in the
Variant Set RST Entry
STATUS; ! Status returned by recursive call

! Push the address of the current Variant Set RST Entry on the Variant
Stack maintained by this routine and GET_RECORD_COMPONENT. This is
how we keep track of nested Variant Sets in the record.

IF .VARSTK INDEX GEQ VARSTK SIZE THEN SIGNAL(DBG\$_VARNESDEP);

VARSTK INDEX = .VARSTK INDEX + 1;

VARSTACK[.VARSTK INDEX - 1] = .VARSETPTR;

! Loop through all the variants in this Variant Set. For each variant in

03FC 00000 GET_RECORD VARIANT:				WORD	Save R2,R3,R4,R5,R6,R7,R8,R9	
59	00000000	EF	9E 00002	MOVAB	VARSTK_INDEX, R9	1035
14		69	D1 00009	CMPL	VARSTK_INDEX, #20	1040
		0D	19 0000C	BLSS	1\$	
	00028A32	8F	DD 0000E	PUSHL	#166450	
00000000G	00	01	FB 00014	CALLS	#1, LIB\$SIGNAL	
		69	D6 0001B 1\$:	INCL	VARSTK_INDEX	1040
		69	DD 0001D	MOVL	VARSTK_INDEX, R0	1040
	04	AC	DD 00020	MOVL	VARSETPTR, R7	

FFOC C940		57	D0	00024	MOVL	R7, VARSTACK1-4[R0]	...	1041
53	1B	A7	9E	0002A	MOVAB	24(R7), VARSETTBL	...	1042
56		01	CE	0002E	MNEGL	#1, I	...	1042
		44	11	00031	BRB	68	...	1041
52		6346	D0	00033	28:	MOVL (VARSETTBL)[I], VARPTR	...	1041
51		69	D0	00037	MOVL	VARSTK_INDEX, R1	...	1041
FF5C C941		52	D0	0003A	MOVL	VARPTR, VARSTACK2-4[R1]	...	1042
54	0B	A2	9E	00040	MOVAB	8(R2), COMPLST	...	1042
55		01	CE	00044	MNEGL	#1, J	...	1042
		29	11	00047	BRB	58	...	1042
58		6445	D0	00049	38:	MOVL (COMPLST)[J], COMPPTR	...	1042
51		69	D0	0004D	MOVL	VARSTK_INDEX, R1	...	1042
AC A941	01	A5	9E	00050	MOVAB	1(R5), VARSTACK3-4[R1]	...	1042
0B AC		58	D1	00056	CMPL	COMPPTR, SYMID	...	1042
		12	13	0005A	BEQL	48	...	1043
0B	14	A8	91	0005C	CMPL	20(COMPPTR), #11	...	1043
		10	12	00060	BNEQ	58	...	1043
	0B	AC	DD	00062	PUSHL	SYMID	...	1043
		58	DD	00065	PUSHL	COMPPTR	...	1043
95 AF		02	FB	00067	CALLS	#2, GET_RECORD_VARIANT	...	1043
04		50	E9	0006B	BLBC	STATUS, 58	...	1043
50		01	D0	0006E	48:	MOVL #1, R0	...	1043
		04	00071		RET		...	1042
D2	55	04	A2	F2	58:	A0BLSS 4(VARPTR), J, 38	...	1041
B7	56	0B	A7	F2	68:	A0BLSS 8(R7), I, 28	...	1044
		69	D7	0007C	DECL	VARSTK_INDEX	...	1044
		50	D4	0007E	CLRL	R0	...	1044
		04	00080		RET		...	1044

; Routine Size: 129 bytes, Routine Base: DBG\$CODE + 30E1


```

:10357
:10358
:10359
:10360
:10361
:10362
:10363
:10364
:10365
:10366
:10367
:10368
:10369
:10370
:10371
:10372
:10373
:10374
:10375
:10376
:10377
:10378
:10379
:10380
:10381
:10382
:10383
:10384
:10385
:10386
:10387
:10388
:10389
:10390
:10391
:10392
:10393
:10394
:10395
:10396
:10397
:10398
:10399
:10400
:10401
:10402
:10403
:10404
:10405
:10406
:10407
:10408
:10409
:10410
:10411
:10412
:10413

```

ROUTINE GET_SET_CONSTANT =

FUNCTION

This routine picks up set constants. It calls DBG\$EXPRESSION_PARSER to parse and evaluate each set constants expression. It also checks the data type of each set constants and converts it to the appropriate data type of the first set constant as necessary.

This routine assumes that the opening set parenthesis has already been found and that the parse pointer points to the start of the first set constant expression. When this routine returns, the parse pointer is left pointing at the first character after the closing set parenthesis.

INPUTS

None.

OUTPUTS

Pointer to set constant value descriptor.

BEGIN

LOCAL

CREATE,

LOW_RANGE_VAL,

SAVED_RADIX,

SETVAL: REF BITVECTOR[],

SETVALPTR: REF DBG\$VALDESC,

THIS_SUBSCR_IS_RANGE,

TYPEID: REF RST\$ENTRY,

VALADDR: REF VECTOR[.LONG],

VALPTR: REF DBG\$VALDESC;

Flag set to true to create set constant value descriptor

Low value of a subscript range

Temporarily saved expression radix

A vector of 256 bits

Pointer to Value Descriptor for declared subscript data type

Flag set if the current subscript is given as a subscript range

Pointer to RST entry

Pointer to integer subscript value

Pointer to subscript Value Descriptor

! Check for Empty Set []. Create a value descriptor for it, mark 'FFFF' in class and dtype fields to indicate this is [], so DBG\$EVAL_LANG_OPERATOR can play with it.

IF .CHARPTR[0] EQL ']'

THEN

BEGIN

SETVALPTR = DBG\$MAKE_SKELETON_DESC(DBG\$K_VALUE_DESC, 8*4);

SETVALPTR[DBG\$B_DHDR_LANG] = 'X'FF';

SETVALPTR[DBG\$B_DHDR_KIND] = RST\$K_DATA;

SETVALPTR[DBG\$B_DHDR_FCODE] = RST\$K_TYPE_SET;

SETVALPTR[DBG\$B_DHDR_VALUE_POINTER] = SETVALPTR[DBG\$B_DHDR_VALUE_ADDRESS];

SETVALPTR[DBG\$B_DHDR_VALUE_CLASS] = 'X'FF';

SETVALPTR[DBG\$B_DHDR_VALUE_DTYPE] = 'X'FF';

SETVALPTR[DBG\$B_DHDR_VALUE_LENGTH] = 32;

CHARPTR = .CHARPTR + 1;

RETURN .SETVALPTR;

END;

```
:10414 10506
:10415 10507
:10416 10508
:10417 10509
:10418 10510
:10419 10511
:10420 10512
:10421 10513
:10422 10514
:10423 10515
:10424 10516
:10425 10517
:10426 10518
:10427 10519
:10428 10520
:10429 10521
:10430 10522
:10431 10523
:10432 10524
:10433 10525
:10434 10526
:10435 10527
:10436 10528
:10437 10529
:10438 10530
:10439 10531
:10440 10532
:10441 10533
:10442 10534
:10443 10535
:10444 10536
:10445 10537
:10446 10538
:10447 10539
:10448 10540
:10449 10541
:10450 10542
:10451 10543
:10452 10544
:10453 10545
:10454 10546
:10455 10547
:10456 10548
:10457 10549
:10458 10550
:10459 10551
:10460 10552
:10461 10553
:10462 10554
:10463 10555
:10464 10556
:10465 10557
:10466 10558
:10467 10559
:10468 10560
:10469 10561
:10470 10562
```

```
: Loop through the set constant expressions for this set constant. Each
: set constant is parsed, evaluated, and converted to the appropriate type
: of the first set constant (with the type being checked in the process).
: Note that TERMINATOR_CODE is set within the loop as a side-effect of the
: call on DBG$EXPRESSION_PARSER.
```

```
CREATE = TRUE;
THIS SUBSCR IS RANGE = FALSE;
TERMINATOR_CODE = TOKEN$K_TERM_COMMA;
WHILE .TERMINATOR_CODE NEQ TOKEN$K_TERM_CLOSE DO
  BEGIN
```

```
: Call the expression parser to pick up the next set constant expression
: and its value. Note that we set the radix to decimal over this call
: and then restore it. Also note that the Expression Parser sets
: TERMINATOR_CODE and TERMINATOR_LENGTH as a side-effect.
```

```
SAVED_RADIX = .EXPRESSION_RADIX;
EXPRESSION_RADIX = DBG$K_DECIMAL;
VALPTR = DBG$EXPRESSION_PARSER (FALSE, SET_CONSTANT_TERM_TBL);
EXPRESSION_RADIX = .SAVED_RADIX;
```

```
: Check the terminator code. If there was no terminator (the input
: line just ended), signal an error. Otherwise we got a comma or clos-
: ing subscript parenthesis and we increment CHARPTR to get past it.
```

```
IF .TERMINATOR_CODE EQL TOKEN$K_TERM_NONE THEN SIGNAL(DBG$MISCLOSUB);
CHARPTR = .CHARPTR + .TERMINATOR_LENGTH;
```

```
: Create a set constant value descriptor. Its type is the type of
: the first set constant data type.
```

```
IF .CREATE
THEN
  BEGIN
    SETVALPTR = DBG$MAKE_SKELETON_DESC(DBG$K_VALUE_DESC, 8*4);
    SETVALPTR[DBG$B_DHDR_LANG] = .DBG$GB_LANGUAGE;
    SETVALPTR[DBG$B_DHDR_KIND] = RST$K_DATA;
    SETVALPTR[DBG$B_DHDR_FCODE] = RST$K_TYPE_SET;
    SETVALPTR[DBG$B_VALUE_LENGTH] = 32;
    SETVALPTR[DBG$B_VALUE_POINTER] = SETVALPTR[DBG$B_VALUE_ADDRESS];
    SETVAL = .SETVALPTR[DBG$B_VALUE_POINTER];
    SELECTONE .VALPTR[DBG$B_DHDR_FCODE] OF
      SET
        [RST$K_TYPE_ATOMIC, RST$K_TYPE_DESCR]:
          BEGIN
            SETVALPTR[DBG$B_DHDR_TYPEID] = DBG$TYPEID_FOR_SET(
              .VALPTR[DBG$B_VALUE_DTYPE], RST$K_TYPE_SET, 256, TRUE);
          END;
        [RST$K_TYPE_ENUM]:
          BEGIN
```

:10471	10563	5
:10472	10564	5
:10473	10565	5
:10474	10566	5
:10475	10567	5
:10476	10568	5
:10477	10569	5
:10478	10570	5
:10479	10571	5
:10480	10572	5
:10481	10573	5
:10482	10574	5
:10483	10575	5
:10484	10576	5
:10485	10577	5
:10486	10578	5
:10487	10579	5
:10488	10580	5
:10489	10581	5
:10490	10582	5
:10491	10583	5
:10492	10584	5
:10493	10585	5
:10494	10586	5
:10495	10587	5
:10496	10588	5
:10497	10589	5
:10498	10590	5
:10499	10591	5
:10500	10592	5
:10501	10593	5
:10502	10594	5
:10503	10595	5
:10504	10596	5
:10505	10597	5
:10506	10598	5
:10507	10599	5
:10508	10600	5
:10509	10601	5
:10510	10602	5
:10511	10603	5
:10512	10604	5
:10513	10605	5
:10514	10606	5
:10515	10607	5
:10516	10608	5
:10517	10609	5
:10518	10610	5
:10519	10611	5
:10520	10612	5
:10521	10613	5
:10522	10614	5
:10523	10615	5
:10524	10616	5
:10525	10617	5
:10526	10618	5
:10527	10619	5

```
TYPEID = VALPTR(DBG$L_DHDR_TYPEID);
SETVALPTR(DBG$L_DHDR_TYPEID) = DBG$TYPEID_FOR_SET(
    .TYPEID[RST$L_DSTPTR], RST$K_TYPE_SET, 258);
END;

[OTHERWISE]:
    SIGNAL(DBG$_ILLTYPE);

    TES;

    CREATE = FALSE;
    END;

! Check that the type of the set item is consistent with the
! type of the set as a whole.
IF NOT DBG$PERFORM TYPEID CHECK (ORT$K_TYPEID_SET_SET,
    .VALPTR, .SETVALPTR)
THEN
    SIGNAL (DBG$_ILLSETCON);

VALADDR = .VALPTR(DBG$$_VALUE_POINTER);

! If the terminator at the end of this set constant expression was a dot
! we have a subrange (for example, '(1..5, 8)'). We thus set
! the subrange-range flag and save the low value of the range, i.e.
! the value we just picked up.
IF .TERMINATOR_CODE EQL TOKEN$K_TERM_DOT
THEN
    BEGIN
        IF .THIS_SUBSCR_IS_RANGE THEN SIGNAL(DBG$_INVRANSPEC);
        THIS_SUBSCR_IS_RANGE = TRUE;
        LOW_RANGE_VAL = .VALADDR[0];
    END

! The terminator was not a dot, so we now have the full set constant
! specification.
ELSE
    BEGIN

        ! If this set constant is specified as a subrange, check that
        ! the first value in the range is not greater than the second.
        ! Also clear the subscript-is-range flag for the next subscript.
        IF .THIS_SUBSCR_IS_RANGE
        THEN
            BEGIN
                IF .LOW_RANGE_VAL GTR .VALADDR[0] THEN SIGNAL(DBG$_INVRANSPEC);
                THIS_SUBSCR_IS_RANGE = FALSE;
            END
```


END:

L1:10616

OFFC 00000 GET_SET_CONSTANT:						
5D	8F	00000000'	FF 91 00002	.WORD	Save R2,R3,R4,R5,R6,R7,R8,R9,R10,R11	1044
			30 12 0000A	CMPB	@CHARPTR, #93	1049
			20 DD 0000C	BNEQ	2\$	
	7E	7A	8F 9A 0000E	PUSHL	#32	1049
00000000G	00		02 FB 00012	MOVZBL	#122, -(SP)	
	52		50 D0 00019	CALLS	#2, DBG\$MAKE_SKELETON_DESC	
03	A2		01 8E 0001C	MOVL	R0, SETVALPTR	
06	A2	0608	8F B0 00020	MNEGB	#1, 3(SETVALPTR)	1049
18	A2	20	A2 9E 00026	MOVW	#1544, 6(SETVALPTR)	1049
14	A2	FFFF0020	8F D0 0002B	MOVAB	32(R2), 24(SETVALPTR)	1049
		00000000'	EF D6 00033	MOVL	#-65504, 20(SETVALPTR)	1050
			0181 31 00039	INCL	CHARPTR	1050
	59		01 D0 0003C	BRW	22\$	1050
			58 D4 0003F	MOVL	#1, CREATE	1051
00000000'	EF		01 D0 00041	CLRL	THIS_SUBSCR_IS_RANGE	1051
	02	00000000'	EF D1 00048	MOVL	#1, TERMINATOR_CODE	1051
			E8 13 0004F	CMPL	TERMINATOR_CODE, #2	1051
	5B	00000000'	EF D0 00051	BEQL	1\$	
				MOVL	EXPRESSION_RADIX, SAVED_RADIX	1052

00000000'	EF	00000000'	0A	D0	00058	MOVL	#10, EXPRESSION RADIX	1052
			EF	9F	0005F	PUSHAB	SET_CONSTANT_TERM_TBL	1052
			7E	D4	00065	CLRL	-(SP)	
D42A	CF		02	FB	00067	CALLS	#2, DBG\$EXPRESSION_PARSER	
00000000'	53		50	D0	0006C	MOVL	R0, VALPTR	
	EF	00000000'	5B	D0	0006F	MOVL	SAVED_RADIX, EXPRESSION_RADIX	1052
			EF	D5	00076	TSTL	TERMINATOR_CODE	1053
			0D	12	0007C	BNEQ	4\$	
		00028E90	8F	DD	0007E	PUSHL	#167568	
00000000G	00		01	FB	00084	CALLS	#1, LIB\$SIGNAL	
00000000'	EF	00000000'	EF	C0	0008B	ADDL2	TERMINATOR_LENGTH, CHARPTR	1053
	7E		59	E9	00096	BLBC	CREATE, 9\$	1054
			20	DD	00099	PUSHL	#32	1054
	7E	7A	8F	9A	0009B	MOVZBL	#122, -(SP)	
00000000G	00		02	FB	0009F	CALLS	#2, DBG\$MAKE_SKELETON_DESC	
	52		50	D0	000A6	MOVL	R0, SETVALPTR	
03	A2	00000000G	00	90	000A9	MOVB	DBG\$GB_LANGUAGE, 3(SETVALPTR)	1054
06	A2	0608	8F	B0	000B1	MOVW	#1544, 6(SETVALPTR)	1054
14	A2		20	B0	000B7	MOVW	#32, 20(SETVALPTR)	1055
18	A2	20	A2	9E	000BB	MOVAB	32(R2), 24(SETVALPTR)	1055
	5A	18	A2	D0	000C0	MOVL	24(SETVALPTR), SETVAL	1055
	50	06	A3	9A	000C4	MOVZBL	6(VALPTR), R0	1055
	02		50	91	000C8	CMPB	R0, #2	1055
			1B	1F	000CB	BLSSU	5\$	
	03		50	91	000CD	CMPB	R0, #3	
			16	1A	000D0	BGTRU	5\$	
	7E	0100	01	DD	000D2	PUSHL	#1	1055
			8F	3C	000D4	MOVZWL	#256, -(SP)	
			08	DD	000D9	PUSHL	#8	
	7E	16	A3	9A	000DB	MOVZBL	22(VALPTR), -(SP)	1055
00000000G	00		04	FB	000DF	CALLS	#4, DBG\$TYPEID_FOR_SET	
			1A	11	000E6	BRB	6\$	
	04		50	91	000E8	CMPB	R0, #4	1056
			1B	12	000EB	BNEQ	7\$	
	56	08	A3	D0	000ED	MOVL	8(VALPTR), TYPEID	1056
	7E	0100	8F	3C	000F1	MOVZWL	#256, -(SP)	1056
			08	DD	000F6	PUSHL	#8	
		0C	A6	DD	000F8	PUSHL	12(TYPEID)	1056
00000000G	00		03	FB	000FB	CALLS	#3, DBG\$TYPEID_FOR_SET	
08	A2		50	D0	00102	MOVL	R0, 8(SETVALPTR)	
			0D	11	00106	BRB	8\$	1055
		000287D8	8F	DD	00108	PUSHL	#165848	1056
00000000G	00		01	FB	0010E	CALLS	#1, LIB\$SIGNAL	
			59	D4	00115	CLRL	CREATE	1057
			52	DD	00117	PUSHL	SETVALPTR	1058
			53	DD	00119	PUSHL	VALPTR	
			02	DD	0011B	PUSHL	#2	1058
00000000G	00		03	FB	0011D	CALLS	#3, DBG\$PERFORM_TYPEID_CHECK	
	0D		50	E8	00124	BLBS	R0, 10\$	
		00028F20	8F	DD	00127	PUSHL	#167712	1058
00000000G	00		01	FB	0012D	CALLS	#1, LIB\$SIGNAL	
	57	18	A3	D0	00134	MOVL	24(VALPTR), VALADDR	1058
	0F	00000000'	EF	D1	00138	CMPB	TERMINATOR_CODE, #15	1059
			15	12	0013F	BNEQ	12\$	
	0D		58	E9	00141	BLBC	THIS_SUBSCR_IS_RANGE, 11\$	1059
		00028F08	8F	DD	00144	PUSHL	#167688	
00000000G	00		01	FB	0014A	CALLS	#1, LIB\$SIGNAL	

58	01	D0	00151	118:	MOVL	#1, THIS_SUBSCR_IS_RANGE	1059
19	1C	11	00154		BRB		1059
54	58	E9	00156	128:	BLBC	SUBSCR_IS_RANGE, 148	1061
54	67	D0	00159		MOVL	(VALADDR), R4	1061
	55	D1	0015C		CMPL	LOW_RANGE_VAL, R4	
	0D	15	0015F		BLEQ		
00000000G	00	8F	DD	00161	PUSHL	#167688	
		01	FB	00167	CALLS	#1, LIB\$SIGNAL	
		58	D4	0016E	CLRL	THIS_SUBSCR_IS_RANGE	1061
		06	11	00170	BRB		1061
54	67	D0	00172	148:	MOVL	(VALADDR), R4	1062
55	54	D0	00175		MOVL	R4, LOW_RANGE_VAL	
	55	D5	00178	158:	TSTL	LOW_RANGE_VAL	1063
	09	19	0017A		BLSS		
00000100	8F	55	D1	0017C	CMPL	LOW_RANGE_VAL, #256	
		0D	15	00183	BLEQ		
00000000G	00	8F	DD	00185	PUSHL	#164424	1063
		01	FB	0018B	CALLS	#1, LIB\$SIGNAL	
		54	D5	00192	TSTL	R4	1063
	09	19	00194		BLSS		
00000100	8F	54	D1	00196	CMPL	R4, #256	
		0D	15	0019D	BLEQ		
00000000G	00	8F	DD	0019F	PUSHL	#164424	1063
		01	FB	001A5	CALLS	#1, LIB\$SIGNAL	
	50	FF	A5	9E	MOVAB	-1(R5), I	1064
		04	11	001B0	BRB		
00	6A	50	E2	001B2	BBSS	I, (SETVAL), 218	
F8	50	54	F3	001B6	AOBLEQ	R4, I, 208	
		FE8B	31	001BA	BRW		1051
	50	52	D0	001BD	MOVL	SETVALPTR, R0	1064
		04	001C0		RET		1064

; Routine Size: 449 bytes, Routine Base: DBG\$CODE + 3162

```

:10559      10650      1
:10560      10651      1
:10561      10652      1
:10562      10653      1
:10563      10654      1
:10564      10655      1
:10565      10656      1
:10566      10657      1
:10567      10658      1
:10568      10659      1
:10569      10660      1
:10570      10661      1
:10571      10662      1
:10572      10663      1
:10573      10664      1
:10574      10665      1
:10575      10666      1
:10576      10667      1
:10577      10668      1
:10578      10669      1
:10579      10670      1
:10580      10671      1
:10581      10672      1
:10582      10673      1
:10583      10674      1
:10584      10675      1
:10585      10676      1
:10586      10677      1
:10587      10678      1
:10588      10679      1
:10589      10680      1
:10590      10681      1
:10591      10682      1
:10592      10683      1
:10593      10684      1
:10594      10685      1
:10595      10686      1
:10596      10687      2
:10597      10688      2
:10598      10689      2
:10599      10690      2
:10600      10691      2
:10601      10692      2
:10602      10693      2
:10603      10694      2
:10604      10695      2
:10605      10696      2
:10606      10697      2
:10607      10698      2
:10608      10699      2
:10609      10700      2
:10610      10701      2
:10611      10702      2
:10612      10703      2
:10613      10704      2
:10614      10705      2
:10615      10706      2

```

ROUTINE GET_SUBSCRIPTS(PRIMPTR): NOVALUE =

FUNCTION

This routine picks up subscript values in an array reference. It calls DBG\$EXPRESSION_PARSER to parse and evaluate each subscript expression. It also checks the data type of each subscript value and converts it to the appropriate data type as necessary. The ultimate subscript values are stored as integers in the Primary Descriptor Array Sub-Node for the array being subscripted. A new Sub-Node for the array element type is then appended so that the Primary Descriptor becomes a descriptor for the array element selected by the subscripting.

This routine also handles subscript ranges, such as ARR(1:5,3:10). It does so by modifying the subscript lower and upper bounds in the Array Sub-Node subscript vector to in effect define a new array, namely the array "slice" defined by the subscript ranges. In this case no Sub-Node for the array element is appended since the Primary Descriptor still defines an array.

This routine assumes that the opening subscript parenthesis has already been found and that the parse pointer points to the start of the first subscript expression. When this routine returns, the parse pointer is left pointing at the first character after the closing subscript parenthesis.

INPUTS

PRIMPTR - A pointer to the Primary Descriptor for an array about to be subscripted.

OUTPUTS

The PRIMPTR Primary Descriptor is changed to include the subscript information (the actual subscript values) and a new Sub-Node for the selected array element. PRIMPTR itself is not changed, however.

BEGIN

MAP

PRIMPTR: REF DBG\$PRIMARY; ! Pointer to array Primary Descriptor

LOCAL

BITSIZE, ! Bit size of subscript value data type
CHECK_VAL,
DECLTYPE: REF DBG\$VALDESC, ! Pointer to Value Descriptor for
declared subscript data type
DESCR: DBG\$STG_DESC, ! String descriptor
DSCADDR: REF DBG\$STG_DESC, ! Pointer to a string descriptor
FCODE, ! Data type FCODE for array element type
LA_PTR: REF VECTOR[.BYTE], ! Lookahead pointer into input
LOW_RANGE_VAL, ! Low value of a subscript range
NODEPTR: REF DBG\$PRIM_NODE, ! Pointer to Prim Descr Array Sub-Node
SAVED_RADIX, ! Temporarily saved expression radix
SUBSCR_COUNT, ! Actual subscript count in input line
SUBVECTOR: ! Pointer to subscript block-vector
REF DBG\$PRIM_NODE_SUBS, ! in Primary Descr Array Sub-Node

```
:10616      10707      2      THIS_SUBSCR_IS_RANGE,      ! Flag set if the current subscript is
:10617      10708      !                               given as a subscript range
:10618      10709      TOKEN,      ! Lexical Token
:10619      10710      TYPECODE,      ! VAX standard type code for atomic type
:10620      10711      TYPEID: REF RST$ENTRY,      ! Holds a typeid
:10621      10712      VALADDR: REF VECTOR[ LONG],      ! Pointer to integer subscript value
:10622      10713      VALPTR: REF DBG$VALDESC;      ! Pointer to subscript Value Descriptor
:10623      10714
:10624      10715      DBG$GL_CURRENT_PRIMARY = .PRIMPTR;
:10625      10716
:10626      10717      ! Check that the Primary Descriptor is really a Primary Descriptor!
:10627      10718
:10628      10719      ! IF .PRIMPTR[DBG$B_DHDR_TYPE] NEQ DBG$K_PRIMARY_DESC
:10629      10720      THEN
:10630      10721      SIGNAL(DBG$_NOTARRAY);
:10631      10722
:10632      10723
:10633      10724
:10634      10725      ! Check that the Primary Descriptor is for an array--otherwise subscripting
:10635      10726      is not allowed.
:10636      10727
:10637      10728      ! IF .PRIMPTR[DBG$B_DHDR_FCODE] NEQ RST$K_TYPE_ARRAY
:10638      10729      THEN
:10639      10730      BEGIN
:10640      10731
:10641      10732      ! Check for the possibility of a substring reference.
:10642      10733      ! E.g., in FORTRAN, if a variable is declared CHARACTER*n
:10643      10734      ! we see it in the DST as an atomic item of type T.
:10644      10735      ! We want to allow X(i:j) in this case.
:10645      10736
:10646      10737      ! IF .PRIMPTR[DBG$B_DHDR_FCODE] EQL RST$K_TYPE_ATOMIC
:10647      10738      THEN
:10648      10739      BEGIN
:10649      10740      DBG$STA_TYP_ATOMIC (.PRIMPTR[DBG$L_DHDR_TYPEID], TYPECODE, BITSIZE);
:10650      10741
:10651      10742
:10652      10743      ! Special case for BASIC. In this case, each element of an
:10653      10744      ! array is a VMS descriptor.
:10654      10745
:10655      10746      ! IF .TYPECODE EQL DSC$K_DTYPE_DSC
:10656      10747      THEN
:10657      10748      BEGIN
:10658      10749      DBG$PRIM_TO_VAL (.PRIMPTR, DBG$K_VALUE_DESC, VALPTR);
:10659      10750      DSCADDR = VALPTR[DBG$A_VALUE_VMSDESC];
:10660      10751      IF .DSCADDR[DSC$B_DTYPE] EQL DSC$K_DTYPE_T
:10661      10752      THEN
:10662      10753      BEGIN
:10663      10754      GET SUBSTRING (.PRIMPTR, .DSCADDR);
:10664      10755      RETURN;
:10665      10756      END;
:10666      10757      END;
:10667      10758
:10668      10759      ! IF .TYPECODE EQL DSC$K_DTYPE_T
:10669      10760      THEN
:10670      10761      BEGIN
:10671      10762      DESCR[DSC$B_CLASS] = DSC$K_CLASS_S;
:10672      10763      DESCR[DSC$B_DTYPE] = DSC$K_DTYPE_T;
```



```
:10673      10764      S      DESCR[DSC$W_LENGTH] = .BITSIZE/8;
:10674      10765      S      GET SUBSTRING (.PRIMPTR, DESCR);
:10675      10766      S      RETURN;
:10676      10767      S      END;
:10677      10768      S      END;
:10678      10769      S
:10679      10770      S      IF .PRIMPTR[DBG$B_DHDR_FCODE] EQL RST$K_TYPE_DESCR
:10680      10771      S      THEN
:10681      10772      S      BEGIN
:10682      10773      S      DBG$STA_TYP_DESCR (.PRIMPTR [DBG$L_DHDR_TYPEID], DSCADDR);
:10683      10774      S      IF .DSCADDR[DSC$B_DTYPE] EQL DSC$K_DTYPE_T
:10684      10775      S      THEN
:10685      10776      S      BEGIN
:10686      10777      S      GET SUBSTRING (.PRIMPTR, .DSCADDR);
:10687      10778      S      RETURN;
:10688      10779      S      END;
:10689      10780      S      END;
:10690      10781      S
:10691      10782      S
:10692      10783      S      ! For typed pointers there are two special cases:
:10693      10784      S      ! For language C, subscripting of pointers is allowed; e.g.,
:10694      10785      S      ! PTR[n] is equivalent to *(PTR+n)
:10695      10786      S      ! For ADA, the pointer dereference is implicit; thus if PTR
:10696      10787      S      ! is a pointer then PTR(I) in ADA is equivalent to PTR^[I]
:10697      10788      S      ! in PASCAL.
:10698      10789      S
:10699      10790      S      IF .PRIMPTR[DBG$B_DHDR_FCODE] EQL RST$K_TYPE_TPTR
:10700      10791      S      THEN
:10701      10792      S      BEGIN
:10702      10793      S      CASE .DBG$GB_LANGUAGE FROM DBG$K_MIN_LANGUAGE TO DBG$K_MAX_LANGUAGE OF
:10703      10794      S      SET
:10704      10795      S      ! For language C, subscripting of pointers is allowed; e.g.,
:10705      10796      S      ! PTR[n] is equivalent to *(PTR+n)
:10706      10797      S      !
:10707      10798      S      [DBG$K_C]:
:10708      10799      S      BEGIN
:10709      10800      S      LOCAL
:10710      10801      S      ADDRESS,                ! Address of array (value of pointer)
:10711      10802      S      DUMMY,                  ! Dummy output parameter
:10712      10803      S      VALPTR: REF DBG$VALDESC; ! Pointer to a Value Descriptor
:10713      10804      S
:10714      10805      S
:10715      10806      S
:10716      10807      S      BUILTIN
:10717      10808      S      REMQUE;
:10718      10809      S
:10719      10810      S
:10720      10811      S      ! Compute the value of the pointer that is represented
:10721      10812      S      ! by the current primary.
:10722      10813      S
:10723      10814      S      DBG$PRIM_TO_VAL (.PRIMPTR, DBG$K_VALUE_DESC, VALPTR);
:10724      10815      S      ADDRESS = .VALPTR[DBG$L_VALUE_VALUE0];
:10725      10816      S
:10726      10817      S
:10727      10818      S      ! Unlink the existing subnode and build a new one to
:10728      10819      S      ! describe an array.
:10729      10820      S
```

```
:10730
:10731
:10732
:10733
:10734
:10735
:10736
:10737
:10738
:10739
:10740
:10741
:10742
:10743
:10744
:10745
:10746
:10747
:10748
:10749
:10750
:10751
:10752
:10753
:10754
:10755
:10756
:10757
:10758
:10759
:10760
:10761
:10762
:10763
:10764
:10765
:10766
:10767
:10768
:10769
:10770
:10771
:10772
:10773
:10774
:10775
:10776
:10777
:10778
:10779
:10780
:10781
:10782
:10783
:10784
:10785
:10786
```

```
10821
10822
10823
10824
10825
10826
10827
10828
10829
10830
10831
10832
10833
10834
10835
10836
10837
10838
10839
10840
10841
10842
10843
10844
10845
10846
10847
10848
10849
10850
10851
10852
10853
10854
10855
10856
10857
10858
10859
10860
10861
10862
10863
10864
10865
10866
10867
10868
10869
10870
10871
10872
10873
10874
10875
10876
10877
```

```
NODEPTR = .PRIMPTR[DBG$$_PRIM_BLINK];
REMQE(.NODEPTR, DUMMY);
TYPEID = DBG$$_TYPEID_FOR_ARRAY(.PRIMPTR[DBG$$_DHDR_TYPEID],
                                .ADDRESS);
DBG$$_BUILD_PRIMARY_SUBNODE (.PRIMPTR, RST$$_DATA, 0,
                              RST$$_TYPE_ARRAY, .TYPEID, 0);
NODEPTR = .PRIMPTR[DBG$$_PRIM_BLINK];
NODEPTR[DBG$$_PNODE_RELOC] = .ADDRESS;
END;
```

```
! For ADA, the pointer dereference is implicit; thus if PTR
! is a pointer then PTR(I) in ADA is equivalent to PTR^[I]
! in PASCAL.
```

```
[DBG$$_ADA]:
BEGIN
```

```
! Dereference the pointer.
```

```
WHILE .PRIMPTR[DBG$$_DHDR_FCODE] EQL RST$$_TYPE_TPTR DO
  GET_DEREFERENCE(.PRIMPTR);
```

```
! If the Primary is still not an array then
! signal an error.
```

```
IF .PRIMPTR[DBG$$_DHDR_FCODE] NEQ RST$$_TYPE_ARRAY
THEN
  SIGNAL(DBG$$_NOTARRAY);
END;
```

```
! For other languages, we cannot subscript typed pointers.
```

```
[INRANGE]:
  SIGNAL(DBG$$_NOTARRAY);
```

```
! We do not expect any other language codes.
```

```
[OUTRANGE]:
  $DBG_ERROR('DBGPARSER\GET_SUBSCRIPTS 5');
```

```
TES;
END
```

```
! Else the variable is neither a string nor a typed pointer
! nor an array so subscripting is not allowed.
```

```
ELSE
  SIGNAL(DBG$$_NOTARRAY);
END;
```

```
10787      10878      | Set up pointers to the Primary Descriptor Array Sub-Node and to the
10788      10879      | subscript vector within that node.
10789      10880
10790      10881      NODEPTR = .PRIMPTR[DBG$L PRIM BLINK];
10791      10882      SUBVECTOR = NODEPTR[DBG$A PNARR SVECTOR];
10792      10883      IF .NODEPTR[DBG$B_PNODE_FCODE] NEQ RST$K_TYPE_ARRAY
10793      10884      THEN
10794      10885          $DBG_ERROR('DBGPARSER\GET_SUBSCRIPTS 10');
10795      10886
10796      10887
10797      10888      | Loop through the subscript expressions for this array reference. Each
10798      10889      | subscript is parsed, evaluated, and converted to the appropriate type
10799      10890      | (with the type being checked in the process). It is then checked for
10800      10891      | being in range and its integer value is stored in the Array Sub-Node's
10801      10892      | subscript block-vector. Note that TERMINATOR_CODE is set within the
10802      10893      | loop as a side-effect of the call on DBG$EXPRESSION_PARSER.
10803      10894
10804      10895      THIS_SUBSCR_IS_RANGE = FALSE;
10805      10896      SUBSCR_COUNT = .NODEPTR[DBG$B PNARR SUBCNT];
10806      10897      TERMINATOR_CODE = TOKEN$K_TERM_COMMA;
10807      10898      WHILE .TERMINATOR_CODE NEQ TOKEN$K_TERM_CLOSE DO
10808      10899          BEGIN
10809      10900
10810      10901
10811      10902          | Check that the actual subscript count does not exceed the dimension
10812      10903          | count for the array.
10813      10904
10814      10905          IF .SUBSCR_COUNT GEQ .NODEPTR[DBG$B_PNARR_DIMCNT]
10815      10906          THEN
10816      10907              SIGNAL(DBG$TOOMANSUB, 1, .NODEPTR[DBG$B_PNARR_DIMCNT]);
10817      10908
10818      10909
10819      10910          | Look for the asterisk. X(*) is the same as X(lower:upper).
10820      10911          | If we find the asterisk then advance the character pointer beyond
10821      10912          | the asterisk and also increment the subscript count.
10822      10913
10823      10914          LA_PTR = .CHARPTR;
10824      10915          WHILE .LA_PTR[0] EQL ' ' DO LA_PTR = .LA_PTR + 1;
10825      10916          IF .LA_PTR[0] EQL '*'
10826      10917          THEN
10827      10918              BEGIN
10828      10919                  CHARPTR = .LA_PTR + 1;
10829      10920
10830      10921
10831      10922          | Call the Lexical Scanner to take us past the ',' or
10832      10923          | or ']' or ')'. This will set TERMINATOR_CODE to the
10833      10924          | terminator that is seen. If we do not see a terminator
10834      10925          | then signal a syntax error. Also signal an error if
10835      10926          | ':' was the terminator.
10836      10927
10837      10928          TOKEN = DBG$LEXICAL_SCANNER (FALSE, FALSE,
10838      10929              .SUBSCRIPT_TERM_TBL, 0);
10839      10930          IF .TOKEN NEQ TERMINATOR_TOKEN
10840      10931          THEN
10841      10932              BEGIN
10842      10933                  LOCAL
10843      10934                      ASCII_STRING: VECTOR[2,BYTE];
```

10844 10935 5
10845 10936 5
10846 10937 5
10847 10938 4
10848 10939 4
10849 10940 4
10850 10941 4
10851 10942 4
10852 10943 4
10853 10944 4
10854 10945 4
10855 10946 4
10856 10947 4
10857 10948 4
10858 10949 4
10859 10950 4
10860 10951 4
10861 10952 4
10862 10953 4
10863 10954 5
10864 10955 5
10865 10956 5
10866 10957 6
10867 10958 6
10868 10959 6
10869 10960 6
10870 10961 6
10871 10962 5
10872 10963 5
10873 10964 4
10874 10965 4
10875 10966 4
10876 10967 4
10877 10968 4
10878 10969 5
10879 10970 4
10880 10971 4
10881 10972 4
10882 10973 4
10883 10974 4
10884 10975 4
10885 10976 4
10886 10977 4
10887 10978 4
10888 10979 4
10889 10980 4
10890 10981 4
10891 10982 4
10892 10983 4
10893 10984 4
10894 10985 4
10895 10986 4
10896 10987 4
10897 10988 4
10898 10989 4
10899 10990 4
10900 10991 4

```
ASCIC_STRING[0] = 1;
ASCIC_STRING[1] = .CHARPTR[0];
SIGNAL(DBG$_SYNERREXPR, 1, ASCIC_STRING);
END;
IF .TERMINATOR_CODE EQL TOKEN$K_TERM_COLON
THEN
    SIGNAL(DBG$_INVRANSPEC);
IF .TERMINATOR_CODE EQL TOKEN$K_TERM_NONE
THEN
    SIGNAL(DBG$_MISCLOSUB);
CHARPTR = .CHARPTR + .TERMINATOR_LENGTH;

! Turn this reference into a range.
! If it was not already a range, then turn all previous
! subscripts into ranges.
IF NOT .NODEPTR[DBG$V_PNARR_RANGE]
THEN
    BEGIN
        NODEPTR[DBG$V_PNARR_RANGE] = TRUE;
        INCR I FROM 0 TO .SUBSCR_COUNT - 1 DO
            BEGIN
                SUBVECTOR[I, DBG$L_PNSUB_LBOUND] =
                    .SUBVECTOR[I, DBG$L_PNSUB_SVALUE];
                SUBVECTOR[I, DBG$L_PNSUB_UBOUND] =
                    .SUBVECTOR[I, DBG$L_PNSUB_SVALUE];
            END;
        END;
        SUBSCR_COUNT = .SUBSCR_COUNT + 1;
        NODEPTR[DBG$V_PNARR_RANGE] = TRUE;
    END
ELSE
    BEGIN
        ! Call the expression parser to pick up the next subscript expression
        ! and its value. Note that we set the radix to decimal over this call
        ! and then restore it. Also note that the Expression Parser sets
        ! TERMINATOR_CODE and TERMINATOR_LENGTH as a side-effect.
        SAVED_RADIX = .EXPRESSION_RADIX;
        EXPRESSION_RADIX = DBG$K_DECIMAL;
        VALPTR = DBG$EXPRESSION_PARSER(FALSE, .SUBSCRIPT_TERM_TBL);
        EXPRESSION_RADIX = .SAVED_RADIX;

        ! Check the terminator code. If there was no terminator (the input
        ! line just ended), signal an error. Otherwise we got a comma or clos-
        ! ing subscript parenthesis and we increment CHARPTR to get past it.
        IF .TERMINATOR_CODE EQL TOKEN$K_TERM_NONE THEN SIGNAL(DBG$_MISCLOSUB);
        CHARPTR = .CHARPTR + .TERMINATOR_LENGTH;
```



```
:10901      10992      4
:10902      10993      4
:10903      10994      4
:10904      10995      4
:10905      10996      4
:10906      10997      4
:10907      10998      4
:10908      10999      4
:10909      11000      4
:10910      11001      4
:10911      11002      4
:10912      11003      4
:10913      11004      4
:10914      11005      4
:10915      11006      4
:10916      11007      4
:10917      11008      4
:10918      11009      4
:10919      11010      4
:10920      11011      4
:10921      11012      4
:10922      11013      4
:10923      11014      4
:10924      11015      4
:10925      11016      4
:10926      11017      4
:10927      11018      4
:10928      11019      4
:10929      11020      4
:10930      11021      4
:10931      11022      4
:10932      11023      4
:10933      11024      4
:10934      11025      4
:10935      11026      4
:10936      11027      4
:10937      11028      4
:10938      11029      4
:10939      11030      4
:10940      11031      6
:10941      11032      6
:10942      11033      6
:10943      11034      6
:10944      11035      6
:10945      11036      6
:10946      11037      6
:10947      11038      6
:10948      11039      6
:10949      11040      6
:10950      11041      6
:10951      11042      6
:10952      11043      6
:10953      11044      6
:10954      11045      6
:10955      11046      6
:10956      11047      6
:10957      11048      6
```

```
! We now need to convert the subscript to one of the appropriate
! dtype. We need to set up a target descriptor for the conversion
! routine. We allocate a skeleton descriptor and fill in some of
! the fields.
```

```
DECLTYPE = DBG$MAKE_SKELETON_DESC(DBG$SK_VALUE_DESC, 4);
DECLTYPE(DBG$B_DHDR_KIND) = RST$K_DATA;
TYPEID = .SUBVECTOR(.SUBSCR_COUNT, DBG$SL_PNSUB_TYPEID);
DECLTYPE(DBG$SL_DHDR_TYPEID) = .TYPEID;
DECLTYPE(DBG$SL_VALUE_POINTER) = DECLTYPE(DBG$A_VALUE_ADDRESS);
```

```
! We now must fill in the FCODE, CLASS, DTYPE, and LENGTH fields
! of the target descriptor. To do this, we need to look at the
! typeid.
```

```
IF .TYPEID EQL 0
THEN
  BEGIN
```

```
! No typeid available - assume longword integer.
```

```
DECLTYPE(DBG$B_DHDR_FCODE) = RST$K_TYPE_ATOMIC;
DECLTYPE(DBG$B_VALUE_CLASS) = DSC$K_CLASS_S;
DECLTYPE(DBG$B_VALUE_DTYPE) = DSC$K_DTYPE_L;
DECLTYPE(DBG$B_VALUE_LENGTH) = 4;
END
```

```
ELSE
  BEGIN
```

```
! Typeid is available - determine FCODE from it.
```

```
DBG$STA_SYMSIZE(.TYPEID, BITSIZE);
FCODE = .TYPEID[RST$B_FCODE];
DECLTYPE(DBG$B_DHDR_FCODE) = .FCODE;
IF .FCODE EQL RST$K_TYPE_ATOMIC
THEN
  BEGIN
```

```
! Atomic data - determine class, dtype, and length from DST.
```

```
DBG$STA_TYP_ATOMIC(.TYPEID, TYPECODE, BITSIZE);
IF .TYPECODE EQL DST$K_BOOL
THEN
  TYPECODE = DSC$K_DTYPE_TF;
DECLTYPE(DBG$B_VALUE_CLASS) = DBG$MAP_DTYPE_CLASS (
  .TYPECODE, .FALSE);
DECLTYPE(DBG$B_VALUE_DTYPE) = .TYPECODE;
IF .TYPECODE NEQ DSC$K_DTYPE_V
AND .TYPECODE NEQ DSC$K_DTYPE_SV
AND .TYPECODE NEQ DSC$K_DTYPE_VU
AND .TYPECODE NEQ DSC$K_DTYPE_SVU
AND .TYPECODE NEQ DSC$K_DTYPE_TF
```

10958 11049 6
10959 11050 6
10960 11051 6
10961 11052 6
10962 11053 6
10963 11054 6
10964 11055 5
10965 11056 5
10966 11057 6
10967 11058 6
10968 11059 6
10969 11060 6
10970 11061 6
10971 11062 6
10972 11063 6
10973 11064 6
10974 11065 6
10975 11066 6
10976 11067 5
10977 11068 6
10978 11069 6
10979 11070 6
10980 11071 6
10981 11072 6
10982 11073 6
10983 11074 6
10984 11075 6
10985 11076 6
10986 11077 5
10987 11078 4
10988 11079 4
10989 11080 4
10990 11081 4
10991 11082 4
10992 11083 4
10993 11084 4
10994 11085 4
10995 11086 4
10996 11087 4
10997 11088 4
10998 11089 4
10999 11090 4
11000 11091 4
11001 11092 4
11002 11093 4
11003 11094 4
11004 11095 5
11005 11096 4
11006 11097 4
11007 11098 4
11008 11099 4
11009 11100 4
11010 11101 4
11011 11102 5
11012 11103 4
11013 11104 4
11014 11105 4

```
THEN
  DECLTYPE[DBG$W_VALUE_LENGTH] = (.BITSIZE+7)/8
ELSE
  DECLTYPE[DBG$W_VALUE_LENGTH] = .BITSIZE;
END

ELSE IF .FCODE EQL RST$K_TYPE_DESCR
THEN
  BEGIN
    ! Descriptor data - determine class, dtype, and length from DST.
    !
    DBG$STA TYP DESCR (.TYPEID, DECLTYPE[DBG$A_VALUE_VMSDESC]);
    IF .DECLTYPE[DBG$B_VALUE_DTYPE] EQL DST$K_BOOL
    THEN
      DECLTYPE[DBG$B_VALUE_DTYPE] = DSC$K_DTYPE_TF;
    END
  ELSE
    BEGIN
      ! Language-specific fcodes. Here we dummy in the dtype field
      ! with a special code. Determine class and length information
      ! using routines in DBGEVALOP.
      !
      DECLTYPE[DBG$B_VALUE_CLASS] = 0;
      DECLTYPE[DBG$B_VALUE_DTYPE] = 0;
      DECLTYPE[DBG$W_VALUE_LENGTH] = (.BITSIZE+7)/8;
    END;
  END;

  ! Finally call the conversion routine. This routine checks that
  ! the conversion is legal before doing it.
  VALPTR = DBG$EVAL_LANG_OPERATOR(DBG$GL_CONVERT_TOKEN, .VALPTR, .DECLTYPE);

  ! Check that the subscript value is within the array bounds and give an
  ! informational message if not (execution continues after the message).
  ! For ADA, we make sure to check enumeration subscripts by their
  ! position and not by their value.
  VALADDR = .VALPTR[DBG$L_VALUE_POINTER];
  IF (.DBG$GB_LANGUAGE EQL DBG$K_ADA) AND
    (.TYPEID NEQ 0) AND
    (.FCODE EQL RST$K_TYPE_ENUM)
  THEN
    CHECK_VAL = DBG$ENUM_POS(.TYPEID, .VALADDR[0])
  ELSE
    CHECK_VAL = .VALADDR[0];

  IF (.CHECK_VAL LSS .SUBVECTOR[.SUBSCR_COUNT, DBG$L_PNSUB_LBOUND]) OR
    (.CHECK_VAL GTR .SUBVECTOR[.SUBSCR_COUNT, DBG$L_PNSUB_UBOUND])
  THEN
    SIGNAL(DBG$_SUBOUTBND, 4, SUBSCR_COUNT + 1, CHECK_VAL,
      .SUBVECTOR[.SUBSCR_COUNT, DBG$L_PNSUB_LBOUND],
```

:11015	11106	4
:11016	11107	4
:11017	11108	4
:11018	11109	4
:11019	11110	4
:11020	11111	4
:11021	11112	4
:11022	11113	4
:11023	11114	4
:11024	11115	4
:11025	11116	4
:11026	11117	4
:11027	11118	4
:11028	11119	4
:11029	11120	5
:11030	11121	5
:11031	11122	5
:11032	11123	5
:11033	11124	5
:11034	11125	5
:11035	11126	6
:11036	11127	6
:11037	11128	6
:11038	11129	7
:11039	11130	7
:11040	11131	7
:11041	11132	7
:11042	11133	7
:11043	11134	6
:11044	11135	6
:11045	11136	5
:11046	11137	5
:11047	11138	5
:11048	11139	5
:11049	11140	5
:11050	11141	5
:11051	11142	5
:11052	11143	5
:11053	11144	5
:11054	11145	5
:11055	11146	5
:11056	11147	4
:11057	11148	5
:11058	11149	5
:11059	11150	5
:11060	11151	5
:11061	11152	5
:11062	11153	5
:11063	11154	5
:11064	11155	5
:11065	11156	5
:11066	11157	6
:11067	11158	6
:11068	11159	6
:11069	11160	6
:11070	11161	6
:11071	11162	6

```
.SUBVECTOR(.SUBSCR_COUNT, DBG$PNSUB_UBOUND));
```

```
! If the terminator at the end of this subscript expression was a colon
! we have a subscript range (for example, "ARR(1:5,2)"). We thus set
! the subscript-range flag and save the low value of the range, i.e.
! the value we just picked up. If this is the first range in the
! subscript list, we also turn all previous subscripts into ranges by
! setting the lower and upper bound for each such subscript to the
! corresponding subscript value. This in effect defines a new array
! which constitutes a "slice" of the original array.
```

```
IF .TERMINATOR_CODE EQL TOKEN$K_TERM_COLON
THEN
```

```
  BEGIN
    IF .THIS_SUBSCR_IS_RANGE THEN SIGNAL(DBG$_INVRANSPEC);
    THIS_SUBSCR_IS_RANGE = TRUE;
    LOW_RANGE_VAL = .VALADDR[0];
    IF NOT .NODEPTR(DBG$V_PNARR_RANGE)
    THEN
      BEGIN
        NODEPTR(DBG$V_PNARR_RANGE) = TRUE;
        INCR I FROM 0 TO .SUBSCR_COUNT - 1 DO
          BEGIN
            SUBVECTOR[I, DBG$PNSUB_LBOUND] =
              .SUBVECTOR[I, DBG$PNSUB_SVALUE];
            SUBVECTOR[I, DBG$PNSUB_UBOUND] =
              .SUBVECTOR[I, DBG$PNSUB_SVALUE];
          END;
        END;
      END;
    END;
  END
```

```
! The terminator was not a colon, so we now have the full subscript
! specification. Fill the subscript value into the Array Sub-Node's
! subscript vector. Set up the bounds for an array "slice" if this
! or any previous subscript specification in this array reference
! consisted of a subscript range. Also bump the subscript count.
```

```
ELSE
```

```
  BEGIN
```

```
! If this subscript is specified as a subscript range, check that
! the first value in the range is not greater than the second.
! Also clear the subscript-is-range flag for the next subscript.
```

```
  IF .THIS_SUBSCR_IS_RANGE
  THEN
```

```
    BEGIN
      IF .LOW_RANGE_VAL GTR .VALADDR[0] THEN SIGNAL(DBG$_INVRANSPEC);
      THIS_SUBSCR_IS_RANGE = FALSE;
    END
```



```
11072 11163 6
11073 11164 6
11074 11165 6
11075 11166 6
11076 11167 6
11077 11168 6
11078 11169 6
11079 11170 6
11080 11171 6
11081 11172 6
11082 11173 6
11083 11174 6
11084 11175 6
11085 11176 6
11086 11177 6
11087 11178 6
11088 11179 6
11089 11180 6
11090 11181 6
11091 11182 6
11092 11183 6
11093 11184 6
11094 11185 6
11095 11186 6
11096 11187 6
11097 11188 6
11098 11189 6
11099 11190 6
11100 11191 6
11101 11192 6
11102 11193 6
11103 11194 6
11104 11195 6
11105 11196 6
11106 11197 6
11107 11198 6
11108 11199 6
11109 11200 6
11110 11201 6
11111 11202 6
11112 11203 6
11113 11204 6
11114 11205 6
11115 11206 6
11116 11207 6
11117 11208 6
11118 11209 6
11119 11210 6
11120 11211 6
11121 11212 6
11122 11213 6
11123 11214 6
11124 11215 6
11125 11216 6
11126 11217 6
11127 11218 6
11128 11219 6

! Otherwise, set the low range value to be the subscript value.
ELSE
    LOW_RANGE_VAL = .VALADDR[0];

! If this or any previous subscript in this array reference con-
! tained a range specification (as in ARR(5:10)), then we arrange
! the array's lower and upper bounds to reflect the array "slice"
! the user is requesting.
IF .NODEPTR[DBG$V_PNARR_RANGE]
THEN
    BEGIN
        SUBVECTOR[.SUBSCR_COUNT, DBG$L_PNSUB_LBOUND] = .LOW_RANGE_VAL;
        SUBVECTOR[.SUBSCR_COUNT, DBG$L_PNSUB_UBOUND] = .VALADDR[0];
    END;

! Finally fill in the subscript value itself (the start of the
! range), increment the subscript count, and loop.
SUBVECTOR[.SUBSCR_COUNT, DBG$L_PNSUB_SVALUE] = .LOW_RANGE_VAL;
SUBSCR_COUNT = .SUBSCR_COUNT + 1;
END;
END;
! End of WHILE loop over subscripts

! We have picked up all the subscripts within this set of subscript paren-
! theses. Now check that the subscript count is the same as the dimension
! count unless the language allows fewer subscripts (as in Pascal where
! the array reference X[2,3][4] is valid). If fewer subscripts are allowed
! we return now, leaving the Array Sub-Node at the end of the Primary.
NODEPTR[DBG$B_PNARR_SUBCNT] = .SUBSCR_COUNT;
IF .SUBSCR_COUNT NEQ .NODEPTR[DBG$B_PNARR_DIMCNT]
THEN
    BEGIN
        IF .MULTIPLE_SUBSCR THEN RETURN;
        SIGNAL(DBG$_TOFEWSUB, 1, .NODEPTR[DBG$B_PNARR_DIMCNT]);
    END;

! If any subscript range was specified, we leave the Array Sub-Node at the
! end of the Primary even if all subscripts were specified. In other words,
! we still have an array, namely the specified "array slice".
IF .NODEPTR[DBG$V_PNARR_RANGE] THEN RETURN;

! All the subscripts are specified. Now set the EVAL bit in the Array
! Sub-Node to indicate that subscripting actually is being done. Also
! construct a new Sub-Node for the array element type and append it to
! the Primary Descriptor. Then return.
NODEPTR[DBG$V_PNODE_EVAL] = TRUE;
```



```
:11129      11220  2  
:11130      11221  
:11131      11222  
:11132      11223  
:11133      11224  
:11134      11225  1
```

```
FCODE = DBG$STA TYPEFCODE(.NODEPTR[DBG$L_PNARR_CELLTYPE]);  
DBG$BUILD_PRIMARY_SUBNODE(.PRIMPTR, RST&R_DATA, 0,  
                           .FCODE, .NODEPTR[DBG$L_PNARR_CELLTYPE], 0);  
RETURN;  
END;
```

```
                                .PSECT DBG$PLIT,NOWRT, SHR, PIC,0  
SF  54  45  47  5C  52  45  53  52  41  50  47  42  44  1A  03398 P.AYL: .ASCII <26>\DBGPARSER\<92>\GET_SUBSCRIPTS 5\  
SF  54  45  47  5C  52  45  53  52  41  50  47  42  44  1B  033B3 P.AYM: .ASCII <27>\DBGPARSER\<92>\GET_SUBSCRIPTS 10\  
      30  31  20  53  54  50  49  52  43  53  42  55  53  033C2
```

```
                                .PSECT DBG$CODE,NOWRT, SHR, PIC,0  
                                OFFC 00000 GET_SUBSCRIPTS:  
                                .WORD Save R2,R3,R4,R5,R6,R7,R8,R9,R10,R11  
                                MOVAB -64(SP), SP  
                                MOVL PRIMPTR, R2  
                                MOVL R2, DBG$GL_CURRENT_PRIMARY  
                                CMPB 2(R2), #12T  
                                BEQL 1$  
                                PUSHL #165864  
                                CALLS #1, LIB$SIGNAL  
                                MOVAB 4(R2), R3  
                                CMPB 2(R3), #1  
                                BNEQ 2$  
                                BRW 14$  
                                CMPB 2(R3), #2  
                                BNEQ 4$  
                                PUSHAB BITSIZE  
                                PUSHAB TYPECODE  
                                PUSHL 8(R2)  
                                CALLS #3, DBG$STA_TYP_ATOMIC  
                                CMPL TYPECODE, #24  
                                BNEQ 3$  
                                PUSHAB VALPTR  
                                MOVZBL #122, -(SP)  
                                PUSHL R2  
                                CALLS #3, DBG$PRIM_TO_VAL  
                                ADDL3 #20, VALPTR, -DSCADDR  
                                MOVL DSCADDR, R0  
                                CMPB 2(R0), #14  
                                BEQL 5$  
                                CMPL TYPECODE, #14  
                                BNEQ 4$  
                                MOVW #270, DESCR+2  
                                DIVL3 #8, BITSIZE, R0  
                                MOVW R0, DESCR  
                                PUSHAB DESCR  
                                BRB 6$  
                                CMPB 2(R3), #3  
                                1065  
                                1071  
                                1072  
                                1072  
                                1072  
                                1073  
                                1074  
                                1074  
                                1074  
                                1075  
                                1075  
                                1075  
                                1076  
                                1076  
                                1076  
                                1077
```

Address	Disassembly	Comment	Symbol
00000000G	00 50 0E	20 08 20 02	AE A2 02 AE A0 0A 50 52 02
0000V	CF	02	A3 03 0096 00 008E 008E 008E 0079
008E 002D	008E 008E 008E 0079	008E 008E 008E 0079	008E 008E 008E 0079
00000000'	00028362	00	00
00000000G	00	24 7E	7A
00000000G	00 50 53 56 50	24 20 18	04 08
00000000G	00 58	00	00
CC1A	CF	18	06
14	A6	02	06
06	06	02	06
F91E	CF	01	01
0008C	0008E	00091	00094
0009B	0009F	000A3	000A5
000A7	000A9	000AE	000AF
000B3	000B5	000B8	000C0
000C8	000D0		
BNEQ	PUSHAB	PUSHL	CALLS
MOVL	CMPB	BNEQ	PUSHL
PUSHL	CALLS	RET	CMPB
BEQL	BRW	CASEB	.WORD
7\$	DSCADDR	8(R2)	#2, DBG\$STA_TYP_DESCR
10\$	DSCADDR, R0	2(R0), #14	7\$
11\$	R0	R2	#2, GET_SUBSTRING
12\$	2(R3), #6	8\$	13\$
13\$	DBG\$GB_LANGUAGE, #0, #10	13\$-9\$,-	13\$-9\$,-
14\$	13\$-9\$,-	13\$-9\$,-	13\$-9\$,-
15\$	13\$-9\$,-	13\$-9\$,-	13\$-9\$,-
16\$	13\$-9\$,-	13\$-9\$,-	13\$-9\$,-
17\$	13\$-9\$,-	13\$-9\$,-	13\$-9\$,-
18\$	13\$-9\$,-	13\$-9\$,-	13\$-9\$,-
19\$	13\$-9\$,-	13\$-9\$,-	13\$-9\$,-
20\$	13\$-9\$,-	13\$-9\$,-	13\$-9\$,-
21\$	13\$-9\$,-	13\$-9\$,-	13\$-9\$,-
22\$	13\$-9\$,-	13\$-9\$,-	13\$-9\$,-
23\$	13\$-9\$,-	13\$-9\$,-	13\$-9\$,-
24\$	13\$-9\$,-	13\$-9\$,-	13\$-9\$,-
25\$	13\$-9\$,-	13\$-9\$,-	13\$-9\$,-
26\$	13\$-9\$,-	13\$-9\$,-	13\$-9\$,-
27\$	13\$-9\$,-	13\$-9\$,-	13\$-9\$,-
28\$	13\$-9\$,-	13\$-9\$,-	13\$-9\$,-
29\$	13\$-9\$,-	13\$-9\$,-	13\$-9\$,-
30\$	13\$-9\$,-	13\$-9\$,-	13\$-9\$,-
31\$	13\$-9\$,-	13\$-9\$,-	13\$-9\$,-
32\$	13\$-9\$,-	13\$-9\$,-	13\$-9\$,-
33\$	13\$-9\$,-	13\$-9\$,-	13\$-9\$,-
34\$	13\$-9\$,-	13\$-9\$,-	13\$-9\$,-
35\$	13\$-9\$,-	13\$-9\$,-	13\$-9\$,-
36\$	13\$-9\$,-	13\$-9\$,-	13\$-9\$,-
37\$	13\$-9\$,-	13\$-9\$,-	13\$-9\$,-
38\$	13\$-9\$,-	13\$-9\$,-	13\$-9\$,-
39\$	13\$-9\$,-	13\$-9\$,-	13\$-9\$,-
40\$	13\$-9\$,-	13\$-9\$,-	13\$-9\$,-
41\$	13\$-9\$,-	13\$-9\$,-	13\$-9\$,-
42\$	13\$-9\$,-	13\$-9\$,-	13\$-9\$,-
43\$	13\$-9\$,-	13\$-9\$,-	13\$-9\$,-
44\$	13\$-9\$,-	13\$-9\$,-	13\$-9\$,-
45\$	13\$-9\$,-	13\$-9\$,-	13\$-9\$,-
46\$	13\$-9\$,-	13\$-9\$,-	13\$-9\$,-
47\$	13\$-9\$,-	13\$-9\$,-	13\$-9\$,-
48\$	13\$-9\$,-	13\$-9\$,-	13\$-9\$,-
49\$	13\$-9\$,-	13\$-9\$,-	13\$-9\$,-
50\$	13\$-9\$,-	13\$-9\$,-	13\$-9\$,-
51\$	13\$-9\$,-	13\$-9\$,-	13\$-9\$,-
52\$	13\$-9\$,-	13\$-9\$,-	13\$-9\$,-
53\$	13\$-9\$,-	13\$-9\$,-	13\$-9\$,-
54\$	13\$-9\$,-	13\$-9\$,-	13\$-9\$,-
55\$	13\$-9\$,-	13\$-9\$,-	13\$-9\$,-
56\$	13\$-9\$,-	13\$-9\$,-	13\$-9\$,-
57\$	13\$-9\$,-	13\$-9\$,-	13\$-9\$,-
58\$	13\$-9\$,-	13\$-9\$,-	13\$-9\$,-
59\$	13\$-9\$,-	13\$-9\$,-	13\$-9\$,-
60\$	13\$-9\$,-	13\$-9\$,-	13\$-9\$,-
61\$	13\$-9\$,-	13\$-9\$,-	13\$-9\$,-
62\$	13\$-9\$,-	13\$-9\$,-	13\$-9\$,-
63\$	13\$-9\$,-	13\$-9\$,-	13\$-9\$,-
64\$	13\$-9\$,-	13\$-9\$,-	1

				F1	11	00146		BRB	11\$		
	01	02		A3	91	00148	12\$:	CMPB	2(R3), #1		1084
				0D	13	0014C		BEQL	14\$		
		000287E8		8F	DD	0014E	13\$:	PUSHL	#165864		1087
50	00000000G	04	00	01	FB	00154		CALLS	#1, LIB\$SIGNAL		
			AC	18	C1	0015B	14\$:	ADDL3	#24, PRIMPTR, R0		1088
			56	60	D0	00160		MOVL	(R0), NODEPTR		
			52	A6	9E	00163		MOVAB	40(R6), SUBVECTOR		1088
			5A	A6	9E	00167		MOVAB	8(NODEPTR), R10		1088
			01	AA	91	0016B		CMPB	1(R10), #1		
				15	13	0016F		BEQL	15\$		
		00000000'		EF	9F	00171		PUSHAB	P_AYM		1088
				01	DD	00177		PUSHL	#1		
		00028362		8F	DD	00179		PUSHL	#164706		
	00000000G	00		03	FB	0017F		CALLS	#3, LIB\$SIGNAL		
			10	AE	D4	00186	15\$:	CLRL	THIS SUBSCR_IS_RANGE		1089
			59	A6	9A	00189		MOVZBL	31(NODEPTR), SUBSCR_COUNT		1089
	00000000'		EF	01	D0	0018D		MOVL	#1, TERMINATOR_CODE		1089
		00000000'	02	EF	D1	00194	16\$:	CMPL	TERMINATOR_CODE, #2		1089
				03	12	0019B		BNEQ	17\$		
				030B	31	0019D		BRW	51\$		
59	1B	A6	08	00	ED	001A0	17\$:	CMPZV	#0, #8, 27(NODEPTR), SUBSCR_COUNT		1090
				13	14	001A6		BGTR	18\$		
			7E	A6	9A	001A8		MOVZBL	27(NODEPTR), -(SP)		1090
				01	DD	001AC		PUSHL	#1		
		00028EB0		8F	DD	001AE		PUSHL	#167600		
	00000000G	00		03	FB	001B4		CALLS	#3, LIB\$SIGNAL		
			AE	EF	D0	001BB	18\$:	MOVL	CHARPTR, LA_PTR		1091
		00000000'	20	BE	91	001C3	19\$:	CMPB	@LA_PTR, #32		1091
				05	12	001C7		BNEQ	20\$		
				04	AE	001C9		INCL	LA_PTR		
				F5	11	001CC		BRB	19\$		
			2A	BE	91	001CE	20\$:	CMPB	@LA_PTR, #42		1091
				03	13	001D2		BEQL	21\$		
				00AF	31	001D4		BRW	28\$		
	00000000'	EF	04	AE	01	C1	001D7	21\$:	ADDL3	#1, LA_PTR, CHARPTR	1091
				7E	D4	001E0		CLRL	-(SP)		1092
		00000000'		EF	DD	001E2		PUSHL	SUBSCRIPT_TERM_TBL		1092
				7E	7C	001E8		CLRQ	-(SP)		1092
	D58C	CF		04	FB	001EA		CALLS	#4, DBG\$LEXICAL_SCANNER		
	18	AE		50	D0	001EF		MOVL	R0, TOKEN		
			50	EF	9E	001F3		MOVAB	TERMINATOR_TOKEN, R0		1093
		00000000'	50	AE	D1	001FA		CMPL	TOKEN, R0		
				1E	13	001FE		BEQL	22\$		
			28	AE	01	90	00200		MOVB	#1, ASCII_STRING	1093
		00000000'	29	AE	FF	90	00204		MOVB	@CHARPTR, ASCII_STRING+1	1093
				AE	9F	0020C		PUSHAB	ASCII_STRING		1093
				01	DD	0020F		PUSHL	#1		
		000289E2		8F	DD	00211		PUSHL	#166370		
	00000000G	00		03	FB	00217		CALLS	#3, LIB\$SIGNAL		
			03	EF	D1	0021E	22\$:	CMPL	TERMINATOR_CODE, #3		1093
		00000000'		0D	12	00225		BNEQ	23\$		
				8F	DD	00227		PUSHL	#167688		1094
	00000000G	00		01	FB	0022D		CALLS	#1, LIB\$SIGNAL		
				EF	D5	00234	23\$:	TSTL	TERMINATOR_CODE		1094
		00000000'		0D	12	0023A		BNEQ	24\$		
				8F	DD	0023C		PUSHL	#167568		1094

00000000G	00	01	FB	00242	CALLS	#1, LIB\$SIGNAL	1094
00000000'	EF	EF	CO	00249	ADDL2	TERMINATOR_LENGTH, CHARPTR	1095
25	6A	13	EO	00254	BBS	#19, (R10), 27\$	1095
02	AA	08	88	00258	BISB2	#8, 2(R10)	1095
	51	01	CE	0025C	MNEGL	#1, I	1095
		18	11	0025F	BRB	26\$	
50	51	14	C5	00261	MULL3	#20, I, R0	
		08	A042	9F	PUSHAB	8(R0)[SUBVECTOR]	1095
			6042	9F	PUSHAB	(R0)[SUBVECTOR]	
	9E	9E	D0	0026C	MOVL	@(SP)+, @(SP)+	
		0C	A042	9F	PUSHAB	12(R0)[SUBVECTOR]	1096
			6042	9F	PUSHAB	(R0)[SUBVECTOR]	
	9E	9E	D0	00276	MOVL	@(SP)+, @(SP)+	
E4	51	59	F2	00279	AOBLS	SUBSCR_COUNT, I, 25\$	1095
		59	D6	0027D	INCL	SUBSCR_COUNT	1096
02	AA	08	88	0027F	BISB2	#8, 2(R10)	1096
		FF0E	31	00283	BRW	16\$	1091
14	AE	EF	D0	00286	MOVL	EXPRESSION_RADIX, SAVED_RADIX	1097
00000000'	EF	0A	D0	0028E	MOVL	#10, EXPRESSION_RADIX	1097
		EF	DD	00295	PUSHL	SUBSCRIPT_TERM_TBL	1098
		7E	D4	0029B	CLRL	-(SP)	
D033	CF	02	FB	0029D	CALLS	#2, DBG\$EXPRESSION_PARSER	
1C	AE	50	D0	002A2	MOVL	R0, VALPTR	
00000000'	EF	14	AE	002A6	MOVL	SAVED_RADIX, EXPRESSION_RADIX	1098
		00000000'	EF	D5	TSTL	TERMINATOR_CODE	1098
		0D	12	002B4	BNEQ	29\$	
		00028E90	8F	DD	PUSHL	#167568	
00000000G	00	01	FB	002BC	CALLS	#1, LIB\$SIGNAL	1098
00000000'	EF	EF	CO	002C3	ADDL2	TERMINATOR_LENGTH, CHARPTR	1099
		04	DD	002CE	PUSHL	#4	
	7E	7A	8F	9A	MOVZBL	#122, -(SP)	
00000000G	00	02	FB	002D4	CALLS	#2, DBG\$MAKE_SKELETON_DESC	
	54	50	D0	002DB	MOVL	R0, DECLTYPE	
07	A4	06	90	002DE	MOVB	#6, 7(DECLTYPE)	1099
55	59	14	C5	002E2	MULL3	#20, SUBSCR_COUNT, R5	1099
		10	A542	9F	PUSHAB	16(R5)[SUBVECTOR]	
	58	9E	D0	002EA	MOVL	@(SP)+, TYPEID	
08	A4	58	D0	002ED	MOVL	TYPEID, 8(DECLTYPE)	1100
18	A4	20	A4	9E	MOVAB	32(R4), 24(DECLTYPE)	1100
	53	14	A4	9E	MOVAB	20(DECLTYPE), R3	1101
			58	D5	TSTL	TYPEID	1100
		0D	12	002FC	BNEQ	30\$	
06	A4	02	90	002FE	MOVB	#2, 6(DECLTYPE)	1101
	63	8F	D0	00302	MOVL	#17301508, (R3)	1101
		6A	11	00309	BRB	33\$	1100
		2C	AE	9F	PUSHAB	BITSIZE	1102
			58	DD	PUSHL	TYPEID	
00000000G	00	02	FB	00310	CALLS	#2, DBG\$STA_SYMSIZE	
	58	18	A8	9A	MOVZBL	24(TYPEID), FCODE	1102
06	A4	58	90	0031B	MOVB	FCODE, 6(DECLTYPE)	1102
	02	58	D1	0031F	CMPL	FCODE, #2	1103
		53	12	00322	BNEQ	34\$	
		2C	AE	9F	PUSHAB	BITSIZE	1103
		34	AE	9F	PUSHAB	TYPECODE	
			58	DD	PUSHL	TYPEID	
00000000G	00	03	FB	0032C	CALLS	#3, DBG\$STA_TYP_ATOMIC	
0000009E	8F	30	AE	D1	CMPL	TYPECODE, #T58	1103

30	AE	04	12	0033B	BNEQ	31\$	1104	
		28	D0	0033D	MOVL	#40, TYPECODE	1104	
		7E	D4	00341	CLRL	-(SP)	1104	
	57	34	AE	D0	00343	MOVL	TYPECODE, R7	
			57	DD	00347	PUSHL	R7	
00000000G	00		02	FB	00349	CALLS	#2, DBG\$MAP_DTYPE_CLASS	
03	A3		50	90	00350	MOVB	R0, 3(R3)	
02	A3		57	90	00354	MOVB	R7, 2(R3)	
	01		57	D1	00358	CMPL	R7, #1	
			14	13	0035B	BEQL	32\$	
	29		57	D1	0035D	CMPL	R7, #41	
			0F	13	00360	BEQL	32\$	
	22		57	D1	00362	CMPL	R7, #34	
			0A	13	00365	BEQL	32\$	
	2A		57	D1	00367	CMPL	R7, #42	
			05	13	0036A	BEQL	32\$	
	28		57	D1	0036C	CMPL	R7, #40	
			26	12	0036F	BNEQ	36\$	
	63	2C	AE	B0	00371	MOVW	BITSIZE, (R3)	
			2C	11	00375	BRB	37\$	
	03		58	D1	00377	CMPL	FCODE, #3	
			18	12	0037A	BNEQ	35\$	
			53	DD	0037C	PUSHL	R3	
			58	DD	0037E	PUSHL	TYPEID	
00000000G	00		02	FB	00380	CALLS	#2, DBG\$STA_TYP_DESCR	
9E	8F	02	A3	91	00387	CMPL	2(R3), #158	
			15	12	0038C	BNEQ	37\$	
	02	A3	28	90	0038E	MOVB	#40, 2(R3)	
			0F	11	00392	BRB	37\$	
		02	A3	B4	00394	CLRW	2(R3)	
50	2C	AE	07	C1	00397	ADDL3	#7, BITSIZE, R0	
51		50	08	C7	0039C	DIVL3	#8, R0, R1	
	63		51	B0	003A0	MOVW	R1, (R3)	
			54	DD	003A3	PUSHL	DECLTYPE	
		20	AE	DD	003A5	PUSHL	VALPTR	
		00000000	EF	9F	003A8	PUSHAB	DBG\$GL_CONVERT_TOKEN	
00000000G	00		03	FB	003AE	CALLS	#3, DBG\$EVAL_LANG_OPERATOR	
1C	AE		50	D0	003B5	MOVL	R0, VALPTR	
0C	AE	18	A0	D0	003B9	MOVL	24(R0), VALADDR	
	09	00000000G	00	91	003BE	CMPL	DBG\$GB_LANGUAGE, #9	
			1E	12	003C5	BNEQ	38\$	
			58	D5	003C7	TSTL	TYPEID	
			1A	13	003C9	BEQL	38\$	
	04		5B	D1	003CB	CMPL	FCODE, #4	
			15	12	003CE	BNEQ	38\$	
	53	0C	BE	D0	003D0	MOVL	@VALADDR, R3	
			53	DD	003D4	PUSHL	R3	
			58	DD	003D6	PUSHL	TYPEID	
00000000G	00		02	FB	003D8	CALLS	#2, DBG\$ENUM_POS	
08	AE		50	D0	003DF	MOVL	R0, CHECK_VAL	
			08	11	003E3	BRB	39\$	
	53	0C	BE	D0	003E5	MOVL	@VALADDR, R3	
			53	D0	003E9	MOVL	R3, CHECK_VAL	
08	AE		08	A542	9F	003ED	PUSHAB	8(R5)[SUBVECTOR]
		0C	AE	D1	003F1	CMPL	CHECK_VAL, @ (SP)+	
	9E		0A	19	003F5	BLSS	40\$	
		0C	A542	9F	003F7	PUSHAB	12(R5)[SUBVECTOR]	

	9E	0C	AE	D1	003FB	CMPL	CHECK_VAL, @ (SP)+		
			21	15	003FF	BLEQ	41\$		
		0C	A542	9F	00401	PUSHAB	12(R5)[SUBVECTOR]	1110	
			9E	DD	00405	PUSHL	@ (SP)+		
		08	A542	9F	00407	PUSHAB	8(R5)[SUBVECTOR]	1110	
			9E	DD	0040B	PUSHL	@ (SP)+		
		10	AE	DD	0040D	PUSHL	CHECK_VAL	1110	
		01	A9	9F	00410	PUSHAB	1(SUBSCR_COUNT)		
			04	DD	00413	PUSHL	#4		
	00000000G		0002868B	8F	DD	00415	PUSHL	#165515	
		00	06	FB	0041B	CALLS	#6, LIB\$SIGNAL		
		03	00000000'	EF	D1	00422	41\$: CMPL	TERMINATOR_CODE, #3	1111
			43	12	00429	BNEQ	45\$		
		0D	10	AE	E9	0042B	BLBC	THIS_SUBSCR_IS_RANGE, 42\$	1112
			00028F08	8F	DD	0042F	PUSHL	#167688	
	00000000G	00	01	FB	00435	CALLS	#1, LIB\$SIGNAL		
		10	AE	01	D0	0043C	42\$: MOVL	#1, THIS_SUBSCR_IS_RANGE	1112
		6E	53	D0	00440	MOVL	R3, LOW_RANGE_VAL	1112	
61		6A	13	E0	00443	BBS	#19, (R10), 50\$	1112	
	02	AA	08	88	00447	BISB2	#8, 2(R10)	1112	
		51	01	CE	0044B	MNEGL	#1, 1	1112	
			18	11	0044E	BRB	44\$		
50		51	14	C5	00450	43\$: MULL3	#20, 1, R0	1113	
			08	A042	9F	00454	PUSHAB	8(R0)[SUBVECTOR]	1113
			6042	9F	00458	PUSHAB	(R0)[SUBVECTOR]		
		9E	9E	D0	0045B	MOVL	@ (SP)+, @ (SP)+		
			0C	A042	9F	0045E	PUSHAB	12(R0)[SUBVECTOR]	1113
			6042	9F	00462	PUSHAB	(R0)[SUBVECTOR]		
		9E	9E	D0	00465	MOVL	@ (SP)+, @ (SP)+		
E4		51	59	F2	00468	44\$: AOBLSS	SUBSCR_COUNT, 1, 43\$	1112	
			3A	11	0046C	BRB	50\$	1111	
		17	10	AE	E9	0046E	45\$: BLBC	THIS_SUBSCR_IS_RANGE, 47\$	1115
		53	6E	D1	00472	CMPL	LOW_RANGE_VAL, -R3	1115	
			0D	15	00475	BLEQ	46\$		
			00028F08	8F	DD	00477	PUSHL	#167688	
	00000000G	00	01	FB	0047D	CALLS	#1, LIB\$SIGNAL		
			10	AE	D4	00484	46\$: CLRL	THIS_SUBSCR_IS_RANGE	1115
			03	11	00487	BRB	48\$	1115	
		6E	53	D0	00489	47\$: MOVL	R3, LOW_RANGE_VAL	1116	
OF		6A	13	E1	0048C	48\$: BBC	#19, (R10), 49\$	1117	
			08	A542	9F	00490	PUSHAB	8(R5)[SUBVECTOR]	1117
		9E	04	AE	D0	00494	MOVL	LOW_RANGE_VAL, @ (SP)+	
			0C	A542	9F	00498	PUSHAB	12(R5)[SUBVECTOR]	1117
		9E	53	D0	0049C	MOVL	R3, @ (SP)+		
			6542	9F	0049F	49\$: PUSHAB	(R5)[SUBVECTOR]	1118	
		9E	04	AE	D0	004A2	MOVL	LOW_RANGE_VAL, @ (SP)+	
			59	D6	004A6	INCL	SUBSCR_COUNT	1118	
			FCE9	51	004AB	50\$: BRW	16\$	1089	
59			59	90	004AB	51\$: MOVB	SUBSCR_COUNT, 31(NODEPTR)	1119	
	1B	A6	08	00	ED	004AF	CMPZV	#0, #8, 27(NODEPTR), SUBSCR_COUNT	1119
				1A	13	004B5	BEQL	52\$	
		3A	00000000'	EF	E8	004B7	BLBS	MULTIPLE_SUBSCR, 53\$	1120
		7E	1B	A6	9A	004BE	MOVZBL	27(NODEPTR), -(SP)	1120
				01	DD	004C2	PUSHL	#1	
			00028EA0	8F	DD	004C4	PUSHL	#167584	
	00000000G	00	03	FB	004CA	CALLS	#3, LIB\$SIGNAL		
23		6A	13	E0	004D1	52\$: BBS	#19, (R10), 53\$	1121	

DBGPARSER
V04-000

C 9
16-Sep-1984 02:10:13
14-Sep-1984 12:17:30

VAX-11 Bliss-32 V4.0-742
[DEBUG.SRC]DBGPARSER.B32:1

Page 354
(43)

02	AA		01	88	004D5	BISB2	#1, 2(R10)		1121
		24	A6	DD	004D9	PUSHL	36(NODEPTR)	:	1122
00000000G	00		01	FB	004DC	CALLS	#1, DBG\$STA_TYPEFCODE	:	
	5B		50	DD	004E3	MOVL	R0, FCODE	:	
			7E	D4	004E6	CLRL	-(SP)	:	1122
		24	A6	DD	004E8	PUSHL	36(NODEPTR)	:	1122
			5B	DD	004EB	PUSHL	FCODE	:	
	7E		06	7D	004ED	MOVQ	#6, -(SP)	:	1122
		04	AC	DD	004F0	PUSHL	PRIMPTR	:	
C851	CF		06	FB	004F3	CALLS	#6, DBG\$BUILD_PRIMARY_SUBNODE	:	1122
			04	004FB	53\$:	RET		:	

; Routine Size: 1273 bytes, Routine Base: DBG\$CODE + 3323

```
11136 11226 1
11137 11227 1
11138 11228 1
11139 11229 1
11140 11230 1
11141 11231 1
11142 11232 1
11143 11233 1
11144 11234 1
11145 11235 1
11146 11236 1
11147 11237 1
11148 11238 1
11149 11239 1
11150 11240 1
11151 11241 1
11152 11242 1
11153 11243 1
11154 11244 1
11155 11245 1
11156 11246 1
11157 11247 1
11158 11248 1
11159 11249 1
11160 11250 1
11161 11251 1
11162 11252 1
11163 11253 1
11164 11254 1
11165 11255 1
11166 11256 1
11167 11257 1
11168 11258 1
11169 11259 1
11170 11260 2
11171 11261 2
11172 11262 2
11173 11263 2
11174 11264 2
11175 11265 2
11176 11266 2
11177 11267 2
11178 11268 2
11179 11269 2
11180 11270 2
11181 11271 2
11182 11272 2
11183 11273 2
11184 11274 2
11185 11275 2
11186 11276 2
11187 11277 2
11188 11278 2
11189 11279 2
11190 11280 2
11191 11281 2
11192 11282 2
```

ROUTINE GET_SUBSTRING (PRIMPTR, DSCADDR) : NOVALUE =

FUNCTION

This routine picks up a substring reference.
For example, in FORTRAN, if a variable X is declared CHARACTER*n then we want to allow X(i:j), where i and j represent the beginning and ending character positions.
In PASCAL, if a variable X is declared PACKED ARRAY[1..N] OF CHAR then we want to allow subscripting X[i] or ranged subscripting X[i:j] to get at substrings of X.

This routine gets called from the GET_SUBSCRIPTS routine, at the point where we discover that what we have is not an array, but is a string.

The expression parser is called to pick up the first subscript. If the terminator ":" is encountered then the expression parser is called again to pick up the upper bound.

These substring bounds then get translated into the PRIM_OFFSET and PRIM_LENGTH fields of the Primary Descriptor, and the SUBREF flag is lit to indicate that a substring selection has taken place.

INPUTS

PRIMPTR - A pointer to the Primary Descriptor for a string about to be subscripted.
DSCADDR - A pointer to the string descriptor representing the string to be subscripted.

OUTPUTS

The PRIMPTR Primary Descriptor is changed to include the substring information.

BEGIN

MAP

PRIMPTR: REF DBG\$PRIMARY,
DSCADDR: REF DBG\$STG_DESC;

LOCAL

HIGH VALUE, : Subscript value
LOW VALUE, : Subscript value
NODEPTR: REF DBG\$PRIM_NODE, : Pointer to Primary Subnode
SAVED RADIX, : Temporary to save radix
VALPTR: REF DBG\$VALDESC; : Pointer to a Value Descriptor

! Temporarily set the radix to decimal. Then call the Expression Parser
! to pick up the lower string bound.

SAVED_RADIX = .EXPRESSION_RADIX;
EXPRESSION_RADIX = DBG\$K_DECIMAL;
VALPTR = DBG\$EXPRESSION_PARSER (FALSE, .SUBSCRIPT_TERM_TBL);
EXPRESSION_RADIX = .SAVED_RADIX;


```
11193 11283
11194 11284
11195 11285
11196 11286
11197 11287
11198 11288
11199 11289
11200 11290
11201 11291
11202 11292
11203 11293
11204 11294
11205 11295
11206 11296
11207 11297
11208 11298
11209 11299
11210 11300
11211 11301
11212 11302
11213 11303
11214 11304
11215 11305
11216 11306
11217 11307
11218 11308
11219 11309
11220 11310
11221 11311
11222 11312
11223 11313
11224 11314
11225 11315
11226 11316
11227 11317
11228 11318
11229 11319
11230 11320
11231 11321
11232 11322
11233 11323
11234 11324
11235 11325
11236 11326
11237 11327
11238 11328
11239 11329
11240 11330
11241 11331
11242 11332
11243 11333
11244 11334
11245 11335
11246 11336
11247 11337
11248 11338
11249 11339

! Check the terminator code. If there was no terminator, (the
! input line just ended), signal an error. If the terminator was
! a comma, this is also an error - we only allow the colon or
! the closing subscript in this case.
IF .TERMINATOR_CODE EQL TOKEN$K_TERM_NONE
THEN
    SIGNAL (DBG$ MISCLOSUB);
IF .TERMINATOR_CODE EQL TOKEN$K_TERM_COMMA
THEN
    SIGNAL (DBG$ SYNERREXPR, 1, UPLIT BYTE (%ASCIC ','));
CHARPTR = .CHARPTR + .TERMINATOR_LENGTH;

! Convert the subscript value to longword integer.
LOW_VALUE = CONVERT_TO_INTEGER (.VALPTR);

! Check for ":" terminator. This indicates we also have to pick
! up the high value.
IF .TERMINATOR_CODE EQL TOKEN$K_TERM_COLON
THEN
    BEGIN
        ! Pick up another expression for the high value.
        SAVED_RADIX = .EXPRESSION_RADIX;
        EXPRESSION_RADIX = DBG$K_DECIMAL;
        VALPTR = DBG$EXPRESSION_PARSER (FALSE, .SUBSCRIPT_TERM_TBL);
        EXPRESSION_RADIX = .SAVED_RADIX;

        ! Check for any of end-of-line, comma, or colon. These are all
        ! errors here.
        IF .TERMINATOR_CODE EQL TOKEN$K_TERM_NONE
        THEN
            SIGNAL (DBG$ MISCLOSUB);
        IF .TERMINATOR_CODE EQL TOKEN$K_TERM_COMMA
        THEN
            SIGNAL (DBG$ SYNERREXPR, 1, UPLIT BYTE (%ASCIC ','));
        IF .TERMINATOR_CODE EQL TOKEN$K_TERM_COMMA
        THEN
            SIGNAL (DBG$ SYNERREXPR, 1, UPLIT BYTE (%ASCIC ':'));
        CHARPTR = .CHARPTR + .TERMINATOR_LENGTH;

        ! Convert the value descriptor to an integer.
        HIGH_VALUE = CONVERT_TO_INTEGER (.VALPTR);
        END

        ! No high value present - same as low value.
    ELSE
        HIGH_VALUE = .LOW_VALUE;

        ! Signal an error if the range is reversed.
```

```
:11250      11340      2
:11251      11341      2
:11252      11342      2
:11253      11343      2
:11254      11344      2
:11255      11345      2
:11256      11346      2
:11257      11347      2
:11258      11348      2
:11259      11349      2
:11260      11350      2
:11261      11351      2
:11262      11352      2
:11263      11353      2
:11264      11354      2
:11265      11355      2
:11266      11356      2
:11267      11357      2
:11268      11358      2
:11269      11359      2
:11270      11360      2
:11271      11361      2
:11272      11362      2
:11273      11363      2
:11274      11364      1
```

```
!
IF .LOW_VALUE LSS 1
OR .HIGH_VALUE GTR .DSCADDR[DSCSW_LENGTH]
OR .LOW_VALUE GTR .HIGH_VALUE
THEN
    SIGNAL (DBG$SUBSTRING, 3, .LOW_VALUE, .HIGH_VALUE,
           .DSCADDR[DSCSW_LENGTH]);

! Signal an error if the values are too large to fit in a Primary.
IF .LOW_VALUE GTR %X'7FFF'
THEN
    SIGNAL(DBG$ILLOFFSET, 1, .LOW_VALUE);
IF (1+.HIGH_VALUE-.LOW_VALUE) GTR %X'7FFF'
THEN
    SIGNAL(DBG$ILLSUBLEN);

! Modify the primary to indicate the substring information
NODEPTR = .PRIMPTR [DBG$L PRIM BLINK];
PRIMPTR [DBG$V_DHDR_SUBREF] = TRUE;
PRIMPTR [DBG$W PRIM OFFSET] = .LOW_VALUE;
PRIMPTR [DBG$W PRIM LENGTH] = 1 + .HIGH_VALUE - .LOW_VALUE;
NODEPTR [DBG$L_PNODE_RELOC] = -1;
END;
```

.PSECT DBG\$PLIT,NOWRT, SHR, PIC,0

```
2C 01 033CF P.AYN: .ASCII <1>\,\
2C 01 033D1 P.AYO: .ASCII <1>\,\
3A 01 033D3 P.AYP: .ASCII <1>\,\
```

.PSECT DBG\$CODE,NOWRT, SHR, PIC,0

```
00FC 00000 GET_SUBSTRING:
      .WORD      Save R2,R3,R4,R5,R6,R7
57 00000000' EF 9E 00002 MOVAB P.AYN, R7
56 00000000G 00 9E 00009 MOVAB LIB$SIGNAL, R6
55 00000000' EF 9E 00010 MOVAB EXPRESSION_RADIX, R5
52 65 D0 00017 MOVL EXPRESSION_RADIX, SAVED_RADIX
65 0A D0 0001A MOVL #10, EXPRESSION_RADIX
      20 A5 D0 0001D PUSHL SUBSCRIPT_TERM_TBL
      7E D4 00020 CLRL -(SP)
CDB5 CF 02 FB 00022 CALLS #2, DBG$EXPRESSION_PARSER
54 50 D0 00027 MOVL R0, VALPTR
65 52 D0 0002A MOVL SAVED_RADIX, EXPRESSION_RADIX
      28 A5 D5 0002D TSTL TERMINATOR_CODE
      09 12 00030 BNEQ 1$
      00028E90 8F DD 00032 PUSHL #167568
66 01 01 FB 00038 CALLS #1, LIB$SIGNAL
      28 A5 D1 0003B 1$: CMPL TERMINATOR_CODE, #1
      0D 12 0003F BNEQ 2$
      57 DD 00041 PUSHL R7
```

			01	DD	00043	PUSHL	#1		
		000289E2	8F	DD	00045	PUSHL	#166370		
			03	FB	00048	CALLS	#3, LIBSSIGNAL		
FBF4	66		A5	CO	0004E	ADDL2	TERMINATOR_LENGTH, CHARPTR		1129
	C5	2C	54	DD	00054	PUSHL	VALPTR		1129
EC26	CF		01	FB	00056	CALLS	#1, CONVERT TO_INTEGER		
	53		50	DO	00058	MOVL	R0, LOW_VALUE		
	03	28	A5	D1	0005E	CMPL	TERMINATOR_CODE, #3		1130
			5E	12	00062	BNEQ	6\$		
	52		65	DO	00064	MOVL	EXPRESSION_RADIX, SAVED_RADIX		1131
	65		0A	DO	00067	MOVL	#10, EXPRESSION_RADIX		1131
		20	A5	DD	0006A	PUSHL	SUBSCRIPT_TERM_TBL		1131
			7E	D4	0006D	CLRL	-(SP)		
CD68	CF		02	FB	0006F	CALLS	#2, DBG\$EXPRESSION_PARSER		
	54		50	DO	00074	MOVL	R0, VALPTR		
	65		52	DO	00077	MOVL	SAVED_RADIX, EXPRESSION_RADIX		1131
		28	A5	D5	0007A	TSTL	TERMINATOR_CODE		1131
			09	12	0007D	BNEQ	3\$		
		00028E90	8F	DD	0007F	PUSHL	#167568		1132
	66		01	FB	00085	CALLS	#1, LIBSSIGNAL		
	01	28	A5	D1	00088	CMPL	TERMINATOR_CODE, #1		1132
			0E	12	0008C	BNEQ	4\$		
		02	A7	9F	0008E	PUSHAB	P.AYD		1132
			01	DD	00091	PUSHL	#1		
		000289E2	8F	DD	00093	PUSHL	#166370		
	66		03	FB	00099	CALLS	#3, LIBSSIGNAL		
	01	28	A5	D1	0009C	CMPL	TERMINATOR_CODE, #1		1132
			0E	12	000A0	BNEQ	5\$		
		04	A7	9F	000A2	PUSHAB	P.AYP		1132
			01	DD	000A5	PUSHL	#1		
		000289E2	8F	DD	000A7	PUSHL	#166370		
	66		03	FB	000AD	CALLS	#3, LIBSSIGNAL		
FBF4	C5	2C	A5	CO	000B0	ADDL2	TERMINATOR_LENGTH, CHARPTR		1132
			54	DD	000B6	PUSHL	VALPTR		1133
EBC4	CF		01	FB	000B8	CALLS	#1, CONVERT TO_INTEGER		
	52		50	DO	000BD	MOVL	R0, HIGH_VALUE		
			03	11	000C0	BRB	7\$		1130
	52		53	DO	000C2	MOVL	LOW_VALUE, HIGH_VALUE		1133
			53	D5	000C5	TSTL	LOW_VALUE		1134
			0D	15	000C7	BLEQ	8\$		
52		08	00	ED	000C9	CMPZV	#0, #16, @DSCADDR, HIGH_VALUE		1134
			05	19	000CF	BLSS	8\$		
	52		53	D1	000D1	CMPL	LOW_VALUE, HIGH_VALUE		1134
			13	15	000D4	BLEQ	9\$		
	7E	08	BC	3C	000D6	MOVZWL	@DSCADDR, -(SP)		1134
			52	DD	000DA	PUSHL	HIGH_VALUE		1134
			53	DD	000DC	PUSHL	LOW_VALUE		
			03	DD	000DE	PUSHL	#3		
		000280D8	8F	DD	000E0	PUSHL	#164056		
	66		05	FB	000E6	CALLS	#5, LIBSSIGNAL		
00007FFF	8F		53	D1	000E9	CMPL	LOW_VALUE, #32767		1135
			0D	15	000F0	BLEQ	10\$		
			53	DD	000F2	PUSHL	LOW_VALUE		1135
			01	DD	000F4	PUSHL	#1		
		000280F0	8F	DD	000F6	PUSHL	#164080		
	66		03	FB	000FC	CALLS	#3, LIBSSIGNAL		
			52	D6	000FF	INCL	R2		1135

		50	7FFF	C3	9E	00101	MOVAB	32767(R3), R0	
		50		S2	D1	00106	CMPL	R2, R0	
				09	15	00109	BLEQ	11\$	
			000280F8	8F	DD	00108	PUSHL	#164088	1135
		66		01	FB	00111	CALLS	#1, LIB\$SIGNAL	
		50	04	AC	D0	00114	MOVL	PRIMPTR, R0	1135
		51	18	A0	D0	00118	MOVL	24(R0), NODEPTR	
	04	A0		02	88	0011C	BISB2	#2, 4(R0)	1136
	10	A0		S3	B0	00120	MOVW	LOW_VALUE, 16(R0)	1136
12	A0	52		S3	A3	00124	SUBW3	LOW_VALUE, R2, 18(R0)	1136
	14	A1		01	CE	00129	MNEGL	#1, -20(NODEPTR)	1136
				04	00	0012D	RET		1136

; Routine Size: 302 bytes, Routine Base: DBG\$CODE + 381C


```

:11276      11365  1
:11277      11366  1
:11278      11367  1
:11279      11368  1
:11280      11369  1
:11281      11370  1
:11282      11371  1
:11283      11372  1
:11284      11373  1
:11285      11374  1
:11286      11375  1
:11287      11376  1
:11288      11377  1
:11289      11378  1
:11290      11379  1
:11291      11380  1
:11292      11381  1
:11293      11382  1
:11294      11383  1
:11295      11384  1
:11296      11385  1
:11297      11386  1
:11298      11387  1
:11299      11388  1
:11300      11389  1
:11301      11390  1
:11302      11391  1
:11303      11392  1
:11304      11393  1
:11305      11394  1
:11306      11395  1
:11307      11396  1
:11308      11397  1
:11309      11398  2
:11310      11399  2
:11311      11400  2
:11312      11401  2
:11313      11402  2
:11314      11403  2
:11315      11404  2
:11316      11405  2
:11317      11406  2
:11318      11407  2
:11319      11408  2
:11320      11409  2
:11321      11410  2
:11322      11411  2
:11323      11412  2
:11324      11413  2
:11325      11414  2
:11326      11415  2
:11327      11416  1

```

ROUTINE OPERATOR_TO_RESTORE_RADIX =

FUNCTION

This routine returns the Operator Lexical Token Entry for the operator which will restore the currently set expression radix. It is used in the processing of the lexical operators %DEC, %HEX, %OCT, and %BIN. If the current radix is decimal, for example, when the %HEX operator is encountered, then the decimal radix must be restored when the range of the %HEX operator ends. Consider this example:

10 + %HEX (20 + 30) - 40

Here 10 and 40 are interpreted as decimal numbers while 20 and 30 are treated as hexadecimal numbers. When the %HEX operator is encountered, this routine is called and returns the %DEC operator. The %DEC operator is pushed onto the operator stack and the radix is then set to hexadecimal. When the minus sign is encountered, it forces evaluation of the stacked %DEC operator which restores the radix to decimal.

The current radix value is stored in and picked up from the OWN variable EXPRESSION_RADIX.

INPUTS

NONE

OUTPUTS

The return value of this routine is the address of the Operator Lexical Token Entry for the lexical operator which restores the current value of EXPRESSION_RADIX.

BEGIN

! Based on the current radix value, return the lexical operator which will restore that radix value when popped from the operator stack and evaluated.

```

IF .EXPRESSION_RADIX EQL DBG$K_DECIMAL THEN RETURN RADIX_OP_DEC;
IF .EXPRESSION_RADIX EQL DBG$K_HEX     THEN RETURN RADIX_OP_HEX;
IF .EXPRESSION_RADIX EQL DBG$K_OCTAL   THEN RETURN RADIX_OP_OCT;
IF .EXPRESSION_RADIX EQL DBG$K_BINARY  THEN RETURN RADIX_OP_BIN;

```

! Any other current radix value is an internal DEBUG error.

```

$DBG_ERROR('DBGPARSER\OPERATOR_TO_RESTORE_RADIX');
RETURN 0;

```

END;

.PSECT DBG\$PLIT,NOWRT, SHR, PIC,0

52 45 50 4F 5C 52 45 53 52 41 50 47 42 44 23 033D5 P.AYQ: .ASCII \DBGPARSER\<92>\OPERATOR_TO_RESTORE_RADIX
45 52 4F 54 53 45 52 5F 4F 54 5F 52 4F 54 41 033E4
44 41 52 5F 033F3
58 49 033F7 .ASCII \IX\

.PSECT DBG\$CODE, NOWRT, SHR, PIC, 0

0004 00000 OPERATOR_TO_RESTORE_RADIX:

52	00000000'	EF	9E	00002	WORD	Save R2	1136
51	00000000'	EF	D0	00009	MOVAB	RADIX_OP_DEC, R2	1140
0A		51	D1	00010	MOVL	EXPRESSION_RADIX, R1	
		04	12	00013	CMPL	R1, #10	
50		62	9E	00015	BNEQ	1\$	
		04	04	00018	MOVAB	RADIX_OP_DEC, R0	
10		51	D1	00019	RET		
		05	12	0001C	CMPL	R1, #16	1140
50	11	A2	9E	0001E	BNEQ	2\$	
		04	04	00022	MOVAB	RADIX_OP_HEX, R0	
08		51	D1	00023	RET		
		05	12	00026	CMPL	R1, #8	1140
50	22	A2	9E	00028	BNEQ	3\$	
		04	04	0002C	MOVAB	RADIX_OP_OCT, R0	
02		51	D1	0002D	RET		
		05	12	00030	CMPL	R1, #2	1140
50	33	A2	9E	00032	BNEQ	4\$	
		04	04	00036	MOVAB	RADIX_OP_BIN, R0	
	30A9	C2	9F	00037	RET		
		01	DD	0003B	PUSHAB	P.AYQ	1141
	00028362	8F	DD	0003D	PUSHL	#1	
00000000G 00		03	FB	00043	PUSHL	#164706	
		50	D4	0004A	CALLS	#3, LIB\$SIGNAL	
		04	04	0004C	CLRL	R0	1141
					RET		1141

; Routine Size: 77 bytes. Routine Base: DBG\$CODE + 394A

```

:11329 11417 1
:11330 11418 1
:11331 11419 1
:11332 11420 1
:11333 11421 1
:11334 11422 1
:11335 11423 1
:11336 11424 1
:11337 11425 1
:11338 11426 1
:11339 11427 1
:11340 11428 1
:11341 11429 1
:11342 11430 1
:11343 11431 1
:11344 11432 1
:11345 11433 1
:11346 11434 1
:11347 11435 1
:11348 11436 1
:11349 11437 1
:11350 11438 1
:11351 11439 1
:11352 11440 1
:11353 11441 1
:11354 11442 1
:11355 11443 1
:11356 11444 1
:11357 11445 1
:11358 11446 1
:11359 11447 1
:11360 11448 1
:11361 11449 1
:11362 11450 1
:11363 11451 1
:11364 11452 1
:11365 11453 1
:11366 11454 1
:11367 11455 1
:11368 11456 1
:11369 11457 1
:11370 11458 1
:11371 11459 1
:11372 11460 1
:11373 11461 1
:11374 11462 1
:11375 11463 1
:11376 11464 1
:11377 11465 1
:11378 11466 1
:11379 11467 2
:11380 11468 2
:11381 11469 2
:11382 11470 2
:11383 11471 2
:11384 11472 2
:11385 11473 2

```

ROUTINE PATHNAME_TO_PRIMARY(PATHDESC, SUBSCR_DESC, PLIPTR,
SAVED_PATHDESC, ARRAY_FLAG) =

FUNCTION

This routine builds a Primary Descriptor Root Node in temporary memory and returns a pointer to that descriptor. The symbol for which the descriptor is built is identified by a Pathname Descriptor. This routine passes the Pathname Descriptor to the GETSYMBOL routine to get the symbol's SYMID, and then builds the Primary Descriptor from that SYMID. For data items, the symbol's FCODE and TYPEID (which identify the data type) are also retrieved and added to the Primary Descriptor.

After the Root Node has been build, DBG\$BUILD_PRIMARY_SUBNODE is called to build a Primary Descriptor Sub-Node for the symbol. This means that the Primary Descriptor is complete when PATHNAME_TO_PRIMARY returns unless further subscripting or other qualification causes additional Sub-Nodes to be appended later.

INPUTS

PATHDESC - The address of a Pathname Descriptor which defines the symbol for which a Primary Descriptor is to be built.

SUBSCR_DESC - Some languages collect their subscripts as they pick up the Primary, and do not apply them until the end. In this case, SUBSCR_DESC is a data structure containing the saved subscripts.

PLIPTR - In a PL/I expression of the form A->B, we first save away then Primary for 'A'. This Primary is in PLIPTR, and must be appended to the beginning of the Primary we now build.

SAVED_PATHDESC: PTH\$PATHNAME -

This is an area of storage in which we can save away a copy of the pathname. This is used to make pathnames sticky in expressions such as P1\A->B

ARRAY_FLAG - An optional fifth parameter, which, if present, indicates that this routine was called as part of an array subscripting operation. This is used in BASIC, where there may be two variables A, one an array and one not, and it is determined from context which is meant. This flag is just passed along to GETSYMBOL so that it can resolve the ambiguity properly.

OUTPUTS

A Primary Descriptor for the symbol specified by PATHDESCR is built and its address is returned as the routine value.

BEGIN

MAP

PATHDESC: REF PTH\$PATHNAME, ! Pointer to input Pathname Descriptor
SAVED_PATHDESC: REF PTH\$PATHNAME, ! Saved copy of pathname.
SUBSCR_DESC: REF SUBSCR\$DESC; ! Pointer to subscript information

```
:11386  
:11387  
:11388  
:11389  
:11390  
:11391  
:11392  
:11393  
:11394  
:11395  
:11396  
:11397  
:11398  
:11399  
:11400  
:11401  
:11402  
:11403  
:11404  
:11405  
:11406  
:11407  
:11408  
:11409  
:11410  
:11411  
:11412  
:11413  
:11414  
:11415  
:11416  
:11417  
:11418  
:11419  
:11420  
:11421  
:11422  
:11423  
:11424  
:11425  
:11426  
:11427  
:11428  
:11429  
:11430  
:11431  
:11432  
:11433  
:11434  
:11435  
:11436  
:11437  
:11438  
:11439  
:11440  
:11441  
:11442
```

```
BUILTIN  
  ACTUALCOUNT;  
  
LOCAL  
  ARR FLAG,  
  BITSIZE,  
  DESCR: REF DBG$STG_DESC,  
  DUMMY,  
  DTYPE,  
  EXPECTED_SUBS,  
  FCODE,  
  INDEX,  
  KIND,  
  LEN,  
  NODEPTR: REF DBG$PRIM_NODE,  
  PATHSTRING,  
  
  PATHVECTOR1: REF VECTOR[.LONG],  
  PATHVECTOR2: REF VECTOR[.LONG],  
  PICKED_UP_SUBSTRING,  
  PRID: REF PRID$ENTRY,  
  PRIMPTR: REF DBG$PRIMARY,  
  SCOPE,  
  SCOPE_STATE,  
  SUBSCR_INDEX,  
  SUBVECTOR:  
    REF DBG$PRIM_NODE_SUBS,  
  SYMID: REF RST$ENTRY,  
  SYMID1: REF RST$ENTRY,  
  SYMID2: REF RST$ENTRY,  
  SYMID_VECT: VECTOR[DBG$K_PATHNAME_SIZE], ! Vector of saved SYMIDs  
  TOOFEWSUB,  
  
  TYP_COMPLST: REF VECTOR[.LONG],  
  TYPEID: REF RST$ENTRY;  
  
! TRUE if expecting an array.  
! Bit length  
! String Descriptor  
! Unused output parameter  
! A type code  
! Count of expected number of subscripts  
! The data type format code for symbol  
! Index into SYMID list  
! RST symbol kind for current symbol  
! Byte length  
! Pointer to Primary sub-node  
! Pointer to pathname Counted ASCII  
! string--used for messages  
! Pathname vector  
! Pathname vector  
! Flag for when we picked up a substring  
! Pointer to Predefined Identifier Entry  
! Pointer to Primary Descriptor  
! Scope where symbol was looked up  
! Kind of scope for SCOPE  
! Index into SUBSCR_DESC  
! Pointer to subscript vector in  
! Primary Descriptor Sub-node  
! SYMID (Symbol ID) for current symbol  
! scratch SYMID (Symbol ID)  
! scratch SYMID (Symbol ID)  
! Flag saying too few subscripts  
! were supplied.  
  
! The Type ID for the symbol's data type  
  
! If the PLIPTR field is not zero then we already have part of the  
! Primary.  
IF .PLIPTR NEQ 0  
THEN  
  BEGIN  
    PRIMPTR = .PLIPTR;  
    NODEPTR = .PRIMPTR[DBG$L_PRIM_BLINK];  
    NODEPTR[DBG$V_PNODE_EVAL] = TRUE;  
  
    IF .PATHDESC[PTH$B_PATHCNT] GTR 1  
    THEN  
      ! Save away this pathname.  
      CH$MOVE(DBG$K_PATHNAME_SIZE, .PATHDESC, .SAVED_PATHDESC)  
  
  ELSE  
    IF .SAVED_PATHDESC[PTH$B_PATHCNT] GTR 1  
    THEN
```



```
:11443      11531  4      BEGIN
:11444      11532  4      LOCAL
:11445      11533  4      I;
:11446      11534  4
:11447      11535  4      ! Merge the previous pathname into this one.
:11448      11536  4      !
:11449      11537  4      PATHVECTOR1 = PATHDESC[PTH$A_PATHVECTOR];
:11450      11538  4      PATHVECTOR2 = SAVED_PATHDESC[PTH$A_PATHVECTOR];
:11451      11539  4      I = .SAVED_PATHDESC[PTH$B_PATHCNT]-1;
:11452      11540  4      DECR J FROM .PATHDESC[PTH$B_TOTCNT]-1 TO 0 DO
:11453      11541  4          PATHVECTOR1[.I+.J] = .PATHVECTOR1[.J];
:11454      11542  4      DECR J FROM .I-1 TO 0 DO
:11455      11543  4          PATHVECTOR1[.J] = .PATHVECTOR2[.J];
:11456      11544  4      PATHDESC[PTH$B_TOTCNT] = .PATHDESC[PTH$B_TOTCNT] + .1;
:11457      11545  4      PATHDESC[PTH$B_PATHCNT] = .PATHDESC[PTH$B_PATHCNT] + .1;
:11458      11546  4      ! <<< INVOCNUM
:11459      11547  3      END;
:11460      11548  3      END
:11461      11549  3
:11462      11550  2      ELSE
:11463      11551  2      BEGIN
:11464      11552  2
:11465      11553  2      ! Allocate space for the Primary Descriptor and fill in the descriptor
:11466      11554  2      ! header fields and sub-node links.
:11467      11555  2
:11468      11556  2      PRIMPTR = DBG$GET TEMPMEM(DBG$K_PRIMARY_SIZE);
:11469      11557  2      PRIMPTR[DBG$B_DHDR_TYPE] = DBG$K_PRIMARY_DESC;
:11470      11558  2      PRIMPTR[DBG$B_DHDR_LANG] = .DBG$GB_LANGUAGE;
:11471      11559  2      PRIMPTR[DBG$W_DHDR_LENGTH] = DBG$K_PRIMARY_SIZE*%UPVAL;
:11472      11560  2      PRIMPTR[DBG$L_PRIM_FLINK] = PRIMPTR[DBG$A_PRIM_FLINK];
:11473      11561  2      PRIMPTR[DBG$L_PRIM_BLINK] = PRIMPTR[DBG$A_PRIM_FLINK];
:11474      11562  2
:11475      11563  2      ! Save away copy of first pathname that we see.
:11476      11564  2      !
:11477      11565  2      IF .PATHDESC[PTH$B_PATHCNT] GTR 1
:11478      11566  2      THEN
:11479      11567  2          CH$MOVE(DBG$K_PATHNAME_SIZE, .PATHDESC, .SAVED_PATHDESC)
:11480      11568  2      ELSE
:11481      11569  2          SAVED_PATHDESC[PTH$B_PATHCNT] = 0;
:11482      11570  2      END;
:11483      11571  2
:11484      11572  2      DBG$GL_CURRENT_PRIMARY = .PRIMPTR;
:11485      11573  2
:11486      11574  2      ! Call GETSYMBOL to get the KIND and SYMID for the symbol.
:11487      11575  2      ! We pass along the flag saying whether this symbol was seen in
:11488      11576  2      ! a subscripted form - getsymbol can use that to resolve ambiguities
:11489      11577  2      ! in BASIC.
:11490      11578  2      ! This also gives the scope where the symbol
:11491      11579  2      ! was looked up and we save that in the Primary Root Node.
:11492      11580  2      ! Then case on the KIND to decide what to do next.
:11493      11581  2      !
:11494      11582  2      IF .COMPONENTS_IN_PATHNAME
:11495      11583  2      THEN
:11496      11584  2          ARR_FLAG = .SUBSCR_DESC[0, SUBSCR$B_SUBCNT] GTR 0
:11497      11585  2      ELSE
:11498      11586  2          ARR_FLAG = ACTUALCOUNT() GTR 4;
:11499      11587  2      DBG$STA_GETSYMBOL(.PATHDESC, SYMID, KIND, SCOPE_STATE, SCOPE, .ARR_FLAG, FALSE);
```

```
:11500 11588 2
:11501 11589
:11502 11590
:11503 11591
:11504 11592
:11505 11593
:11506 11594
:11507 11595
:11508 11596
:11509 11597
:11510 11598
:11511 11599
:11512 11600
:11513 11601
:11514 11602
:11515 11603
:11516 11604
:11517 11605
:11518 11606
:11519 11607
:11520 11608
:11521 11609
:11522 11610
:11523 11611
:11524 11612
:11525 11613
:11526 11614
:11527 11615
:11528 11616
:11529 11617
:11530 11618
:11531 11619
:11532 11620
:11533 11621
:11534 11622
:11535 11623
:11536 11624
:11537 11625
:11538 11626
:11539 11627
:11540 11628
:11541 11629
:11542 11630
:11543 11631
:11544 11632
:11545 11633
:11546 11634
:11547 11635
:11548 11636
:11549 11637
:11550 11638
:11551 11639
:11552 11640
:11553 11641
:11554 11642
:11555 11643
:11556 11644
```

```
PRIMPTR[DBG$B_PRIM_SCOPE_STATE] = .SCOPE_STATE;
PRIMPTR[DBG$L_PRIM_SCOPE] = .SCOPE;
```

```
SELECTONE .KIND OF
SET
```

```
! Handle the case where the symbol is not found in DEBUG's symbol
! table (i.e., the RST). Signal the appropriate error message.
```

```
[RST$K_INVALID]:
BEGIN
```

```
! First check for Predefined Identifier reserved by the language.
! Note: We do this after we are sure that the symbol is not in
! the symbol table. In PASCAL you may redefine the predefined
! symbols. Currently, PASCAL is the only language where we
! have predefined identifiers. FORTRAN has them but they are
! prefaced by a dot "." and are picked up by now.
! Check that no invocation number is present.
```

```
IF .PATHDESC[PTH$B_LOCINVOC] EQL 0
THEN
```

```
BEGIN
```

```
INCR I FROM 0 TO .PRIDTBL[-1] - 1 DO
```

```
BEGIN
```

```
PRID = .PRIDTBL[I] + TABLEBASE;
```

```
IF .PRID[PRID$B_KIND] EQL PRID$K_CONSTANT
```

```
THEN
```

```
BEGIN
```

```
LOCAL
```

```
TEMP_NAME : REF VECTOR [,BYTE];
```

```
PATHVECTOR1 = PATHDESC[PTH$A_PATHVECTOR];
```

```
TEMP_NAME = .PATHVECTOR1[0];
```

```
IF CR$EQL(.PRID[PRID$B_LENGTH],
```

```
PRID[PRID$A_NAME],
```

```
TEMP_NAME[0],
```

```
TEMP_NAME[1])
```

```
THEN
```

```
BEGIN
```

```
PRIMPTR = CREATE_PRID_CONSTANT(.PRID);
```

```
RETURN .PRIMPTR;
```

```
END;
```

```
END;
```

```
END;
```

```
END;
```

```
IF .DBG$GB_LANGUAGE EQL DBG$K_COBOL
```

```
THEN
```

```
DBG$NCOB_PATHDESC_TO_CS(.PATHDESC, PATHSTRING)
```

```
ELSE
```

```
DBG$NPATHDESC_TO_CS(.PATHDESC, PATHSTRING);
```

```
11557 11645 SIGNAL(DBG$_NOSYMBOL, 1, .PATHSTRING);
11558 11646 END;
11559 11647
11560 11648
11561 11649 ! Handle the case where the symbol is not unique. Signal the appro-
11562 11650 priate error.
11563 11651
11564 11652 [RST$K_NOTUNIQUE,
11565 11653 RST$K_OVERLOAD]:
11566 11654 BEGIN
11567 11655
11568 11656 IF .DBG$GB_LANGUAGE EQL DBG$K_COBOL
11569 11657 THEN
11570 11658     DBG$NCOB_PATHDESC_TO_CS(.PATHDESC, PATHSTRING)
11571 11659 ELSE
11572 11660     DBG$NPATHDESC_TO_CS(.PATHDESC, PATHSTRING);
11573 11661
11574 11662 IF .KIND EQL RST$K_NOTUNIQUE
11575 11663 THEN
11576 11664     SIGNAL(DBG$_NOUNIQUE, 1, .PATHSTRING)
11577 11665 ELSE
11578 11666     SIGNAL(DBG$_NOTUNQOVR, 1, .PATHSTRING);
11579 11667
11580 11668 END;
11581 11669
11582 11670
11583 11671 ! Handle all lexical entities, instruction labels, and line numbers.
11584 11672 ! Zero the FCODE and TYPEID--they do not apply in these cases.
11585 11673
11586 11674 [RST$K_ROUTINE,
11587 11675 RST$K_BLOCK,
11588 11676 RST$K_ENTRY,
11589 11677 RST$K_LABEL,
11590 11678 RST$K_LINE]:
11591 11679 BEGIN
11592 11680
11593 11681     PRIMPTR[DBG$B_DHDR_KIND] = .KIND;
11594 11682     PRIMPTR[DBG$L_DHDR_SYMID] = .SYMID;
11595 11683     FCODE = 0;
11596 11684     TYPEID = 0;
11597 11685
11598 11686
11599 11687 ! Build a Primary Descriptor Sub-Node for the symbol we have so far.
11600 11688
11601 11689     DBG$BUILD_PRIMARY_SUBNODE(.PRIMPTR, .KIND, .SYMID, .FCODE, .TYPEID, 0);
11602 11690 END;
11603 11691
11604 11692
11605 11693 ! Handle data items. Here we get the FCODE and TYPEID of the data
11606 11694 item and store them in the Primary Descriptor.
11607 11695
11608 11696 [RST$K_DATA]:
11609 11697 BEGIN
11610 11698
11611 11699
11612 11700 ! Walk the up-scope chain collecting all data symids above this one..
11613 11701 ! This is in order to handle a case such as "X.Y.Z". In this example,
```

```

:11614
:11615
:11616
:11617
:11618
:11619
:11620
:11621
:11622
:11623
:11624
:11625
:11626
:11627
:11628
:11629
:11630
:11631
:11632
:11633
:11634
:11635
:11636
:11637
:11638
:11639
:11640
:11641
:11642
:11643
:11644
:11645
:11646
:11647
:11648
:11649
:11650
:11651
:11652
:11653
:11654
:11655
:11656
:11657
:11658
:11659
:11660
:11661
:11662
:11663
:11664
:11665
:11666
:11667
:11668
:11669
:11670

```

```

: we build a vector of 3 symids: one for Z, one for Y, and one for X.
INDEX = -1;
WHILE .KIND EQL RST$K_DATA DO
  BEGIN
    INDEX = .INDEX + 1;
    IF .INDEX GEQ DBG$K_PATHNAME_SIZE
      THEN
        $DBG ERROR('DBGPARSER\PATHNAME_TO_PRIMARY symid stack overflow');
    SYMID_VECT[.INDEX] = .SYMID;
    SYMID = .SYMID[RST$L_UPSCOPEPTR];
    KIND = .SYMID[RST$B_KIND];
  END;

: Now walk back down the symid list, building up the Primary Descriptor.
SUBSCR_INDEX = 0;
EXPECTED_SUBS = 0;
TOOFEWSUB = FALSE;
DECR I FROM .INDEX TO 0 DO
  BEGIN
    SYMID = .SYMID_VECT[I];

    : Fill in the root node SYMID if we are at the top of the list.
    IF (.I EQL .INDEX) AND (.PLIPTR EQL 0)
      THEN
        PRIMPTR[DBG$L_DHDR_SYMID0] = .SYMID;

    : Get the KIND, FCODE, and TYPEID, and build a new subnode.
    KIND = .SYMID[RST$B_KIND];
    DBG$STA_SETCONTEXT(.SYMID);
    DBG$STA_SYMTYPE(.SYMID, FCODE, TYPEID);
    DBG$BUI[PRIMARY_SUBNODE(.PRIMPTR, .KIND, .SYMID, .FCODE, .TYPEID, 0)];

    : Obtain a pointer to the newly-built subnode.
    NODEPTR = .PRIMPTR[DBG$L_PRIM_BLINK];

    : If there was an address override as in P->X then set a flag
    : saying not to use the SYMID of X for address computations
    : (we still retain it for printing, however)
    IF (.I EQL .INDEX) AND (.PLIPTR NEQ 0)
      THEN
        IF .FCODE EQL RST$K_TYPE_RECORD
          THEN
            NODEPTR[DBG$L_PNODE_SYMID] = 0
          ELSE
            NODEPTR[DBG$V_PNODE_IGNORE] = TRUE;

    : If we have just attached an array subnode then fill
    : in the subscript information here.
    IF .FCODE EQL RST$K_TYPE_ARRAY

```



```

:11671      11759  4
:11672      11760  4
:11673      11761  5
:11674      11762  5
:11675      11763  5
:11676      11764  5
:11677      11765  5
:11678      11766  5
:11679      11767  6
:11680      11768  6
:11681      11769  6
:11682      11770  6
:11683      11771  6
:11684      11772  7
:11685      11773  7
:11686      11774  7
:11687      11775  6
:11688      11776  7
:11689      11777  7
:11690      11778  7
:11691      11779  7
:11692      11780  7
:11693      11781  7
:11694      11782  7
:11695      11783  7
:11696      11784  7
:11697      11785  7
:11698      11786  6
:11699      11787  6
:11700      11788  6
:11701      11789  6
:11702      11790  6
:11703      11791  6
:11704      11792  6
:11705      11793  6
:11706      11794  6
:11707      11795  7
:11708      11796  7
:11709      11797  7
:11710      11798  7
:11711      11799  7
:11712      11800  6
:11713      11801  6
:11714      11802  6
:11715      11803  6
:11716      11804  6
:11717      11805  6
:11718      11806  6
:11719      11807  6
:11720      11808  7
:11721      11809  7
:11722      11810  7
:11723      11811  7
:11724      11812  7
:11725      11813  7
:11726      11814  8
:11727      11815  7

```

```

AND .COMPONENTS_IN_PATHNAME
THEN
  BEGIN
    SUBVECTOR = NODEPTR(DBG$A_PNARR_SVECTOR);
    EXPECTED_SUBS = .EXPECTED_SUBS + .NODEPTR(DBG$B_PNARR_DIMCNT);
    ! Loop through the dimensions of the array.
    INCR J FROM 0 TO .NODEPTR(DBG$B_PNARR_DIMCNT)-1 DO
      BEGIN
        ! Signal an error if not enough subscripts were supplied.
        IF (.SUBSCR_INDEX GEQ .SUBSCR_DESC[0, SUBSCR$B_SUBCNT])
          AND ((.SUBSCR_DESC[0, SUBSCR$B_SUBCNT] NEQ 0) OR
              (.I NEQ 0))
          THEN
            BEGIN
              ! We have a problem in that we know we don't have
              ! enough subscripts, but we don't know exactly
              ! how many we were expecting. So what we do here
              ! is just set a flag saying to signal the error
              ! later on (after we do know).
              TOOFEWSUB = TRUE;
              SUBSCR_DESC[0, SUBSCR$B_SUBCNT] = 0;
              END;
              ! Special check for no subscripts specified - treat this
              ! the same as if (*,*,....) were specified (aggregate
              ! examine).
              IF .SUBSCR_DESC[0, SUBSCR$B_SUBCNT] EQL 0
                THEN
                  INCR J FROM 0 TO .NODEPTR(DBG$B_PNARR_DIMCNT) DO
                    BEGIN
                      SUBSCR_DESC[.SUBSCR_INDEX + .J,
                        SUBSCR$V_RANGE] = TRUE;
                      SUBSCR_DESC[.SUBSCR_INDEX + .J,
                        SUBSCR$V_aster] = TRUE;
                    END;
                    ! If we were given a range then fix up the sub-node
                    ! to reflect a ranged examine.
                    IF .SUBSCR_DESC[.SUBSCR_INDEX, SUBSCR$V_RANGE]
                      THEN
                        BEGIN
                          ! Do not allow asterisk unless we are at the
                          ! bottom level. *** TEMPORARY
                          IF (.I NEQ 0) AND
                            (.SUBSCR_DESC[0, SUBSCR$B_SUBCNT] NEQ 0)
                            THEN

```

11728	11816	7
11729	11817	7
11730	11818	7
11731	11819	7
11732	11820	7
11733	11821	7
11734	11822	7
11735	11823	7
11736	11824	7
11737	11825	8
11738	11826	8
11739	11827	8
11740	11828	9
11741	11829	9
11742	11830	9
11743	11831	9
11744	11832	9
11745	11833	8
11746	11834	7
11747	11835	7
11748	11836	7
11749	11837	7
11750	11838	8
11751	11839	8
11752	11840	8
11753	11841	8
11754	11842	8
11755	11843	8
11756	11844	8
11757	11845	8
11758	11846	8
11759	11847	8
11760	11848	8
11761	11849	8
11762	11850	8
11763	11851	7
11764	11852	7
11765	11853	7
11766	11854	7
11767	11855	7
11768	11856	7
11769	11857	7
11770	11858	7
11771	11859	6
11772	11860	7
11773	11861	7
11774	11862	7
11775	11863	7
11776	11864	7
11777	11865	7
11778	11866	8
11779	11867	7
11780	11868	7
11781	11869	7
11782	11870	7
11783	11871	7
11784	11872	7

```
IF .SUBSCR_DESC[.SUBSCR_INDEX, SUBSCR$V_ASTER]
THEN
  SIGNAL(DBG$_ILLASTER)
ELSE
  SIGNAL(DBG$_ILLRANGE);

IF NOT .NODEPTR(DBG$V_PNARR_RANGE)
THEN
  BEGIN
    NODEPTR(DBG$V_PNARR_RANGE) = TRUE;
    INCR K FROM 0 TO .J-1 DO
      BEGIN
        SUBVECTOR[.K, DBG$L_PNSUB_LBOUND] =
          .SUBVECTOR[.K, DBG$L_PNSUB_SVALUE];
        SUBVECTOR[.K, DBG$L_PNSUB_UBOUND] =
          .SUBVECTOR[.K, DBG$L_PNSUB_SVALUE];
      END;
    END;

IF NOT .SUBSCR_DESC[.SUBSCR_INDEX, SUBSCR$V_ASTER]
THEN
  BEGIN
    ! Check for reversed range.
    IF .SUBSCR_DESC[.SUBSCR_INDEX, SUBSCR$L_LBOUND] GTR
      .SUBSCR_DESC[.SUBSCR_INDEX, SUBSCR$L_UBOUND]
    THEN
      SIGNAL(DBG$_INVRANSPEC);

    SUBVECTOR[.J, DBG$L_PNSUB_LBOUND] =
      .SUBSCR_DESC[.SUBSCR_INDEX, SUBSCR$L_LBOUND];
    SUBVECTOR[.J, DBG$L_PNSUB_UBOUND] =
      .SUBSCR_DESC[.SUBSCR_INDEX, SUBSCR$L_UBOUND];
    END;

    SUBVECTOR[.J, DBG$L_PNSUB_SVALUE] =
      .SUBVECTOR[.J, DBG$L_PNSUB_LBOUND]
  END

  ! Fill in the subscript value.
ELSE
  BEGIN
    LOCAL
      VAL;

    VAL = .SUBSCR_DESC[.SUBSCR_INDEX, SUBSCR$L_LBOUND];
    IF (.VAL LSS .SUBVECTOR[.J, DBG$L_PNSUB_LBOUND]) OR
      (.VAL GTR .SUBVECTOR[.J, DBG$L_PNSUB_UBOUND])
    THEN
      SIGNAL(DBG$ SUBOUTEND, 4, .SUBSCR_INDEX, .VAL,
        .SUBVECTOR[.J, DBG$L_PNSUB_LBOUND],
        .SUBVECTOR[.J, DBG$L_PNSUB_UBOUND]);
    SUBVECTOR[.J, DBG$L_PNSUB_SVALUE] =
      .SUBSCR_DESC[.SUBSCR_INDEX, SUBSCR$L_LBOUND];
```

```

11785 11873 6
11786 11874 6
11787 11875 6
11788 11876 6
11789 11877 6
11790 11878 6
11791 11879 6
11792 11880 6
11793 11881 7
11794 11882 7
11795 11883 7
11796 11884 7
11797 11885 7
11798 11886 6
11799 11887 6
11800 11888 6
11801 11889 5
11802 11890 5
11803 11891 5
11804 11892 5
11805 11893 5
11806 11894 5
11807 11895 5
11808 11896 5
11809 11897 5
11810 11898 5
11811 11899 6
11812 11900 6
11813 11901 6
11814 11902 6
11815 11903 6
11816 11904 6
11817 11905 6
11818 11906 6
11819 11907 6
11820 11908 6
11821 11909 5
11822 11910 4
11823 11911 4
11824 11912 4
11825 11913 4
11826 11914 4
11827 11915 4
11828 11916 4
11829 11917 4
11830 11918 4
11831 11919 4
11832 11920 5
11833 11921 4
11834 11922 5
11835 11923 5
11836 11924 5
11837 11925 5
11838 11926 5
11839 11927 5
11840 11928 5
11841 11929 5

```

```

END;

! If we previously got a range, then make this dimension
! into a range also.
IF .NODEPTR[DBG$V_PNARR_RANGE] AND
NOT .SUBSCR_DESC[SUBSCR_INDEX, SUBSCR$V_RANGE]
THEN
BEGIN
SUBVECTOR[J, DBG$P_PNSUB_LBOUND] =
SUBVECTOR[J, DBG$P_PNSUB_SVALUE];
SUBVECTOR[J, DBG$P_PNSUB_UBOUND] =
SUBVECTOR[J, DBG$P_PNSUB_SVALUE];
END;

SUBSCR_INDEX = .SUBSCR_INDEX + 1;
END;

! Fill in the field that gives the count of subscripts.
! Also, unless this is an array slice, then light the
! EVAL bit which says that subscripting is being done,
! and tack on a new subnode.
NODEPTR[DBG$B_PNARR_SUBCNT] = .NODEPTR[DBG$B_PNARR_DIMCNT];
IF NOT .NODEPTR[DBG$V_PNARR_RANGE]
THEN
BEGIN
NODEPTR[DBG$V_PNODE_EVAL] = TRUE;

! Attach a new Primary sub-node for the array elements.
TYPEID = .NODEPTR[DBG$P_PNARR_CELLTYPE];
FCODE = DBG$STA_TYPEFCODE(.TYPEID);
DBG$BUILD_PRIMARY_SUBNODE(.PRIMPTR, RST$K_DATA, 0,
FCODE, .TYPEID, 0);
NODEPTR = .PRIMPTR[DBG$P_PRIM_BLINK];
END;
END;

! If we have just attached a record subnode then fill
! in the component information here. Note that this must
! be done after the array case above, to properly handle
! arrays of records.
IF (.FCODE EQL RST$K_TYPE_RECORD) AND
(.COMPONENTS_IN_PATHNAME) AND
(.I NEQ 0)
THEN
BEGIN
! If we are not at the bottom symid, then there must be
! a symid for the record component below this one.
! We need to fill in the corresponding record sub-node
! with the component index. This component index is
! used by MODIFY_PRIMARY to compute logical successor/
! predecessor. We also light the 'EVAL' bit in the

```

```
:11842      11930      S
:11843      11931      S
:11844      11932      S
:11845      11933      S
:11846      11934      S
:11847      11935      S
:11848      11936      6
:11849      11937      6
:11850      11938      6
:11851      11939      6
:11852      11940      6
:11853      11941      6
:11854      11942      6
:11855      11943      6
:11856      11944      6
:11857      11945      7
:11858      11946      7
:11859      11947      7
:11860      11948      6
:11861      11949      6
:11862      11950      6
:11863      11951      S
:11864      11952      4
:11865      11953      4
:11866      11954      4
:11867      11955      4
:11868      11956      4
:11869      11957      4
:11870      11958      4
:11871      11959      4
:11872      11960      4
:11873      11961      4
:11874      11962      4
:11875      11963      4
:11876      11964      4
:11877      11965      4
:11878      11966      4
:11879      11967      3
:11880      11968      4
:11881      11969      4
:11882      11970      4
:11883      11971      4
:11884      11972      5
:11885      11973      5
:11886      11974      5
:11887      11975      5
:11888      11976      6
:11889      11977      6
:11890      11978      6
:11891      11979      5
:11892      11980      5
:11893      11981      6
:11894      11982      6
:11895      11983      6
:11896      11984      6
:11897      11985      5
:11898      11986      5
```

```
! sub-node.
NODEPTR[DBG$V_PNODE_EVAL] = TRUE;
SYMID1 = .SYMID_VECT[.I-1];
TYPCOMPLST = TYPEID[RST$L_TYPCOMPLST];
INCR J FROM 0 TO .TYPEID[RST$L_TYPCOMPCNT] - 1 DO
  BEGIN
    SYMID2 = .TYPCOMPLST[.J];

    ! Use the DSTPTR to determine whether we are
    ! really looking at the right component.
    IF .SYMID1[RST$L_DSTPTR] EQL .SYMID2[RST$L_DSTPTR]
    THEN
      BEGIN
        NODEPTR[DBG$W_PNREC_INDEX] = .J + 1;
        EXITLOOP;
      END;

    ! We should not fall through to here.
    END;
  END;
END;

! If we have not exhausted the given list of subscripts,
! then first check for a substring reference.
! We must have seen something that
! looks like a ranged subscript "(i:j)". In order for a
! substring to be legal then the data type must
! either be text or one of the decimal string types.
PICKED_UP_SUBSTRING = 0;
IF (.SUBSCR_INDEX EQL (.SUBSCR_DESC[0, SUBSCR$B_SUBCNT]-1))
AND .SUBSCR_DESC[.SUBSCR_INDEX, SUBSCR$V_RANGE]
AND .SUBSCR_DESC[.SUBSCR_INDEX, SUBSCR$V_MARKER]
AND (NOT .SUBSCR_DESC[.SUBSCR_INDEX, SUBSCR$V_aster])
THEN
  BEGIN
    DTYPE = DSC$K_DTYPE_Z;
    IF .KIND EQL RST$K_DATA
    THEN
      BEGIN
        SELECTONE .FCODE OF
          SET
            [RST$K_TYPE_ATOMIC]:
              BEGIN
                DBG$STA_TYP_ATOMIC(.TYPEID, DTYPE, BITSIZE);
                LEN = .BITSIZE / 8;
              END;
            [RST$K_TYPE_DESCR]:
              BEGIN
                DBG$STA_TYP_DESCR(.TYPEID, DESCR);
                DTYPE = .DESCR[DSC$B_DTYPE];
                LEN = .DESCR[DSC$W_LENGTH];
              END;
            [RST$K_TYPE_PICT, RST$K_TYPE_RECORD]:
```


:11899	11987	6
:11900	11988	6
:11901	11989	6
:11902	11990	6
:11903	11991	5
:11904	11992	5
:11905	11993	5
:11906	11994	5
:11907	11995	5
:11908	11996	5
:11909	11997	5
:11910	11998	5
:11911	11999	6
:11912	12000	5
:11913	12001	6
:11914	12002	6
:11915	12003	6
:11916	12004	6
:11917	12005	6
:11918	12006	6
:11919	12007	6
:11920	12008	6
:11921	12009	6
:11922	12010	6
:11923	12011	6
:11924	12012	6
:11925	12013	6
:11926	12014	6
:11927	12015	8
:11928	12016	8
:11929	12017	6
:11930	12018	7
:11931	12019	7
:11932	12020	6
:11933	12021	6
:11934	12022	6
:11935	12023	6
:11936	12024	6
:11937	12025	6
:11938	12026	6
:11939	12027	6
:11940	12028	6
:11941	12029	6
:11942	12030	6
:11943	12031	6
:11944	12032	6
:11945	12033	6
:11946	12034	6
:11947	12035	6
:11948	12036	6
:11949	12037	6
:11950	12038	5
:11951	12039	5
:11952	12040	5
:11953	12041	5
:11954	12042	5
:11955	12043	5

```
BEGIN
DTYPE = DSC$K_DTYPE_T;
LEN = 10000;
! Pick large number to allow
! arbitrarily long substring.
END;

TES;
IF (.DTYPE EQL DSC$K_DTYPE_T) OR
(.DTYPE EQL DSC$K_DTYPE_NU) OR
(.DTYPE EQL DSC$K_DTYPE_NL) OR
(.DTYPE EQL DSC$K_DTYPE_NLO) OR
(.DTYPE EQL DSC$K_DTYPE_NR) OR
(.DTYPE EQL DSC$K_DTYPE_NRO) OR
(.DTYPE EQL DSC$K_DTYPE_NZ)
THEN
BEGIN
! Modify the primary to indicate the substring information
PRIMPTR [DBG$V_DHDR_AGGR] = FALSE;
PRIMPTR [DBG$V_DHDR_SUBREF] = TRUE;

IF .SUBSCR_DESC[.SUBSCR_INDEX, SUBSCR$L_LBOUND] GTR
'X'7FFFF'
THEN
SIGNAL(DBG$ ILLOFFSET, 1,
.SUBSCR_DESC[.SUBSCR_INDEX, SUBSCR$L_LBOUND]);
PRIMPTR [DBG$W PRIM OFFSET] =
.SUBSCR_DESC[.SUBSCR_INDEX, SUBSCR$L_LBOUND];
IF ((.SUBSCR_DESC[.SUBSCR_INDEX, SUBSCR$L_LBOUND] +
.SUBSCR_DESC[.SUBSCR_INDEX, SUBSCR$L_UBOUND]-1)
GTR .LEN) OR
(.SUBSCR_DESC[.SUBSCR_INDEX, SUBSCR$L_LBOUND]
LSS 1)
THEN
SIGNAL(DBG$ SUBSTRING, 3,
.SUBSCR_DESC[.SUBSCR_INDEX, SUBSCR$L_LBOUND],
.SUBSCR_DESC[.SUBSCR_INDEX, SUBSCR$L_UBOUND],
.LEN);

IF .SUBSCR_DESC[.SUBSCR_INDEX, SUBSCR$L_UBOUND] GTR
'X'7FFF'
THEN
SIGNAL(DBG$ ILLSUBLEN, 1,
.SUBSCR_DESC[.SUBSCR_INDEX, SUBSCR$L_UBOUND]);
PRIMPTR [DBG$W PRIM LENGTH] =
.SUBSCR_DESC[.SUBSCR_INDEX, SUBSCR$L_UBOUND];
NODEPTR = PRIMPTR [DBG$ PRIM BLINK];
NODEPTR [DBG$L PNODE_RELOC] = -1;
PICKED_UP_SUBSTRING = 1;
SUBSCR_INDEX = .SUBSCR_INDEX + 1;
END
ELSE
! We have seen something that looks like a substring
! but the data type is wrong. Catch that case here and
! signal an error.
```

```

:11956 12044 5
:11957 12045 5
:11958 12046 5
:11959 12047 4
:11960 12048 4
:11961 12049 4
:11962 12050 4
:11963 12051 4
:11964 12052 4
:11965 12053 4
:11966 12054 4
:11967 12055 4
:11968 12056 4
:11969 12057 4
:11970 12058 4
:11971 12059 4
:11972 12060 4
:11973 12061 4
:11974 12062 4
:11975 12063 4
:11976 12064 4
:11977 12065 4
:11978 12066 4
:11979 12067 4
:11980 12068 4
:11981 12069 4
:11982 12070 4
:11983 12071 4
:11984 12072 4
:11985 12073 4
:11986 12074 4
:11987 12075 4
:11988 12076 4
:11989 12077 4
:11990 12078 1

```

```

        SIGNAL(DBG$_ILLSUBSTR);
    END

ELSE
    ! We have seen something that looks like a substring
    ! but the data type is wrong. Catch that case here and
    ! signal an error.
    SIGNAL(DBG$_ILLSUBSTR);
END;

! Signal an error if we picked up too many or too few subscripts.
IF .TOOFEWSUB
THEN
    SIGNAL(DBG$ TOOFEWSUB, 1, .EXPECTED SUBS);
IF .SUBSCR_INDEX LSS .SUBSCR_DESC[0, SUBSCR$B_SUBCNT]
THEN
    SIGNAL(DBG$ TOOMANSUB, 1, .SUBSCR_INDEX-.PICKED_UP_SUBSTRING);
END;

! Anything else we treat as an internal DEBUG error.
[OTHERWISE]:
    $DBG_ERROR('DBGPARSER\PATHNAME_TO_PRIMARY 10');
TES;

! Return a pointer to the Primary Descriptor to the caller.
RETURN .PRIMPTR;
END;

```

```

.PSECT DBG$PLIT,NOWRT, SHR, PIC,0
48 54 41 50 5C 52 45 53 52 41 50 47 42 44 32 033F9 P.AYR: .ASCII \2DBGPARSER\<92>\PATHNAME_TO_PRIMARY sym\
59 52 41 4D 49 52 50 5F 4F 54 5F 45 4D 41 4E 03408
6C 66 72 65 76 6F 20 6B 63 61 74 73 20 64 69 03417
48 54 41 50 5C 52 45 53 52 41 50 47 42 44 20 0341B .ASCII \id stack overflow\
59 52 41 4D 49 52 50 5F 4F 54 5F 45 4D 41 4E 0342A
P.AYS: .ASCII \ DBGPARSER\<92>\PATHNAME_TO_PRIMARY 10\
30 31 20 0342C
0343B
0344A

```

```

.PSECT DBG$CODE,NOWRT, SHR, PIC,0
OFFC 00000 PATHNAME_TO_PRIMARY:
5E FEE4 CE 9E 00002 .WORD Save R2,R3,R4,R5,R6,R7,R8,R9,R10,R11 : 1141
58 04 AC D0 00007 .MOVAB -284(SP), SP : 1152
MOVL PATHDESC, R8

```

59	10	AC	D0	0000B	MOVL	SAVED_PATHDESC, R9	1152
	04	AE	D4	0000F	CLRL	4(SP)	1151
	0C	AC	D5	00012	TSTL	PLIPTR	
		50	13	00015	BEQL	5\$	
	04	AE	D6	00017	INCL	4(SP)	
58	0C	AC	D0	0001A	MOVL	PLIPTR, PRIMPTR	1151
56	18	AB	D0	0001E	MOVL	24(PRIMPTR), NODEPTR	1151
0A	A6	01	88	00022	BISB2	#1, 10(NODEPTR)	1151
01	01	A8	91	00026	CMPB	1(R8), #1	1152
		67	1A	0002A	BGTRU	6\$	
01	01	A9	91	0002C	CMPB	1(R9), #1	1152
		6A	1B	00030	BLEQU	8\$	
57	08	A8	9E	00032	MOVAB	8(R8), PATHVECTOR1	1153
52	08	A9	9E	00036	MOVAB	8(R9), PATHVECTOR2	1153
51	01	A9	9A	0003A	MOVZBL	1(R9), 1	1153
		51	D7	0003E	DECL	1	
50		68	9A	00040	MOVZBL	(R8), J	1154
		09	11	00043	BRB	2\$	
53	51	50	C1	00045	ADDL3	J, 1, R3	
6743		6740	D0	00049	MOVL	(PATHVECTOR1)[J], (PATHVECTOR1)[R3]	
F4		50	F4	0004E	SOBGEQ	J, 1\$	
50		51	D0	00051	MOVL	1, J	1154
		05	11	00054	BRB	4\$	
6740		6240	D0	00056	MOVL	(PATHVECTOR2)[J], (PATHVECTOR1)[J]	
F8		50	F4	0005B	SOBGEQ	J, 3\$	
68		51	80	0005E	ADDB2	1, (R8)	1154
01	A8	51	80	00061	ADDB2	1, 1(R8)	1154
		35	11	00065	BRB	8\$	1151
		09	DD	00067	PUSHL	#9	1155
00000000G	00	01	FB	00069	CALLS	#1, DBG\$GET_TEMPMEM	
	5B	50	D0	00070	MOVL	R0, PRIMPTR	
02	AB	79	8F	90	MOVB	#121, 2(PRIMPTR)	1155
03	AB	00000000G	00	90	MOVB	DBG\$GB_LANGUAGE, 3(PRIMPTR)	1155
	6B		24	B0	MOVW	#36, (PRIMPTR)	1155
14	AB	14	AB	9E	MOVAB	20(PRIMPTR), 20(PRIMPTR)	1156
18	AB	14	AB	9E	MOVAB	20(PRIMPTR), 24(PRIMPTR)	1156
	01	01	A8	91	CMPB	1(R8), #1	1156
			06	1B	BLEQU	7\$	
69	68		34	28	MOVCL	#52, (R8), (R8)	1156
			03	11	BRB	8\$	
		01	A9	94	CLRB	1(R9)	1156
00000000G	00		5B	D0	MOVL	PRIMPTR, DBG\$GL_CURRENT_PRIMARY	1157
	0C	00000000'	EF	E9	BLBC	COMPONENTS_IN_PATHNAME, 9\$	1158
			50	D4	CLRL	R0	1158
00	08	BC	08	ED	CMPZV	#8, #8, @SUBSCR_DESC, #0	
			09	14	BGTR	10\$	
			09	11	BRB	11\$	
			50	D4	CLRL	R0	1158
04			6C	91	CMPB	(AP), #4	
			02	1B	BLEQU	11\$	
			50	D6	INCL	R0	
51			50	D0	MOVL	R0, ARR_FLAG	
			7E	D4	CLRL	-(SP)	1158
			51	DD	PUSHL	ARR_FLAG	
			2C	AE	PUSHAB	SCOPE	
			34	AE	PUSHAB	SCOPE_STATE	
			3C	AE	PUSHAB	KIND	

			44	AE	9F	000CF	PUSHAB	SYMID		
				58	DD	000D2	PUSHL	R8		
	00000000G	00		07	FB	000D4	CALLS	#7, DBG\$STA_GETSYMBOL		
		1C	28	AE	90	000DB	MOVB	SCOPE_STATE, 28(PRIMPTR)		1158
		20	24	AE	D0	000E0	MOVL	SCOPE, 32(PRIMPTR)		1158
			2C	AE	D0	000E5	MOVL	KIND, R3		1159
				03	13	000E9	BEQL	12\$		1159
				0083	31	000EB	BRW	18\$		
			02	AB	95	000EE	TSTB	2(R8)		1160
				4E	12	000F1	BNEQ	15\$		
		50		EF	D0	000F3	MOVL	PRIDTBL, R0		1161
		59		AO	D0	000FA	MOVL	-4(R0), R9		
		55		01	CE	000FE	MNEGL	#1, I		
				3A	11	00101	BRB	14\$		
		50		EF	9E	00103	MOVAB	TABLEBASE, R0		1161
54		50		FF	45	0010A	ADDL3	@PRIDTBL[], R0, PRID		
		01		64	91	00113	CMPB	(PRID), #1		1161
				25	12	00116	BNEQ	14\$		
		57	08	AB	9E	0011E	MOVAB	8(R8), PATHVECTOR1		1162
		50		67	D0	0011C	MOVL	(PATHVECTOR1), TEMP_NAME		1162
		52	08	A4	9A	0011F	MOVZBL	8(PRID), R2		1162
		51		60	9A	00123	MOVZBL	(TEMP_NAME), R1		1162
51	00	09	A4	52	2D	00126	CMPC5	R2, 9(PRID), #0, R1, 1(TEMP_NAME)		1162
			01	AO		0012C				
				0D	12	0012E	BNEQ	14\$		
	EA65	CF		54	DD	00130	PUSHL	PRID		1163
		5B		01	FB	00132	CALLS	#1, CREATE PRID_CONSTANT		
				50	D0	00137	MOVL	R0, PRIMPTR		
				050D	31	0013A	BRW	84\$		1163
		55		59	F2	0013D	AOBLSS	R9, I, 13\$		1161
C2		03		00	91	00141	CMPB	DBG\$GB_LANGUAGE, #3		1164
				0E	12	00148	BNEQ	16\$		
			34	AE	9F	0014A	PUSHAB	PATHSTRING		1164
	00000000G	00		58	DD	0014D	PUSHL	R8		
				02	FB	0014F	CALLS	#2, DBG\$NCOB_PATHDESC_TO_CS		
				0C	11	00156	BRB	17\$		
			34	AE	9F	00158	PUSHAB	PATHSTRING		1164
				58	DD	0015B	PUSHL	R8		
	00000000G	00		02	FB	0015D	CALLS	#2, DBG\$NPATHDESC_TO_CS		
			34	AE	DD	00164	PUSHL	PATHSTRING		1164
				01	DD	00167	PUSHL	#1		
				8F	DD	00169	PUSHL	#164344		
				4A	11	0016F	BRB	23\$		
		09		53	D1	00171	CMPL	R3, #9		1165
				05	13	00174	BEQL	19\$		
		0D		53	D1	00176	CMPL	R3, #13		
				43	12	00179	BNEQ	24\$		
		03		00	91	0017B	CMPB	DBG\$GB_LANGUAGE, #3		1165
				0E	12	00182	BNEQ	20\$		
			34	AE	9F	00184	PUSHAB	PATHSTRING		1165
				58	DD	00187	PUSHL	R8		
	00000000G	00		02	FB	00189	CALLS	#2, DBG\$NCOB_PATHDESC_TO_CS		
				0C	11	00190	BRB	21\$		
			34	AE	9F	00192	PUSHAB	PATHSTRING		1166
				58	DD	00195	PUSHL	R8		
	00000000G	00		02	FB	00197	CALLS	#2, DBG\$NPATHDESC_TO_CS		
		09		53	D1	0019E	CMPL	R3, #9		1166

		34	0D	12	001A1	BNEQ	22\$		
			AE	DD	001A3	PUSHL	PATHSTRING		1166
			01	DD	001A6	PUSHL	#1		
		000281F0	8F	DD	001A8	PUSHL	#164336		
			0B	11	001AE	BRB	23\$		
		34	AE	DD	001B0	PUSHL	PATHSTRING		1166
			01	DD	001B3	PUSHL	#1		
		000282A8	8F	DD	001B5	PUSHL	#164520		
			0485	31	001BB	BRW	83\$		
	02		53	D1	001BE	CMPL	R3, #2		1167
			05	19	001C1	BLSS	25\$		
	05		53	D1	001C3	CMPL	R3, #5		
			05	15	001C6	BLEQ	26\$		
	08		53	D1	001C8	CMPL	R3, #8		
			23	12	001CB	BNEQ	27\$		
07	AB		53	90	001CD	MOV8	R3, 7(PRIMPTR)		1168
0C	AB	30	AE	D0	001D1	MOVL	SYMID, 12(PRIMPTR)		1168
		38	AE	7C	001D6	CLRQ	TYPEID		1168
			7E	D4	001D9	CLRL	-(SP)		1168
		3C	AE	DD	001DB	PUSHL	TYPEID		
		44	AE	DD	001DE	PUSHL	FCODE		
		3C	AE	DD	001E1	PUSHL	SYMID		
			53	DD	001E4	PUSHL	R3		
			5B	DD	001E6	PUSHL	PRIMPTR		
C4E8	CF		06	FB	001E8	CALLS	#6, DBG\$BUILD_PRIMARY_SUBNODE		
			045A	31	001ED	BRW	84\$		1159
	06		53	D1	001F0	CMPL	R3, #6		1169
			03	13	001F3	BEQL	28\$		
			043D	31	001F5	BRW	82\$		
	55		01	CE	001F8	MNEGL	#1, INDEX		1170
	52	30	AE	D0	001FB	MOVL	SYMID, R2		1171
	06	2C	AE	D1	001FF	CMPL	KIND, #6		1170
			31	12	00203	BNEQ	31\$		
			55	D6	00205	INCL	INDEX		1170
	34		55	D1	00207	CMPL	INDEX, #52		1170
			15	19	0020A	BLSS	30\$		
		00000000'	EF	9F	0020C	PUSHAB	P.AYR		1171
			01	DD	00212	PUSHL	#1		
		00028362	8F	DD	00214	PUSHL	#164706		
			03	FB	0021A	CALLS	#3, LIB\$SIGNAL		
00000000G	00		52	D0	00221	MOVL	R2, SYMID_VECT[INDEX]		1171
4C	AE45	10	A2	D0	00226	MOVL	16(R2), SYMID		1171
30	AE	30	AE	D0	0022B	MOVL	SYMID, R2		1171
52			A2	9A	0022F	MOVZBL	20(R2), KIND		
2C	AE	14	C9	11	00234	BRB	29\$		1170
			54	D4	00236	CLRL	SUBSCR INDEX		1171
		1C	AE	7C	00238	CLRQ	TOOFEWSUB		1172
	53	01	A5	9E	0023B	MOVAB	1(R5), I		1174
			0274	31	0023F	BRW	64\$		
30	AE	4C	AE43	D0	00242	MOVL	SYMID_VECT[1], SYMID		1172
			59	D4	00248	CLRL	R9		1172
	55		53	D1	0024A	CMPL	I, INDEX		
			0C	12	0024D	BNEQ	33\$		
			59	D6	0024F	INCL	R9		
		0C	AC	D5	00251	TSTL	PLIPTR		
			05	12	00254	BNEQ	33\$		
0C	AB	30	AE	D0	00256	MOVL	SYMID, 12(PRIMPTR)		1173

		57	30	AE	DO	0025B	33\$:	MOVL	SYMID, R7	1173
	2C	AE	14	A7	9A	0025F		MOVZBL	20(R7), KIND	1173
				57	DD	00264		PUSHL	R7	1173
	00000000G	00		01	FB	00266		CALLS	#1, DBG\$STA_SETCONTEXT	1173
			38	AE	9F	0026D		PUSHAB	TYPEID	1173
			40	AE	9F	00270		PUSHAB	FCODE	
				57	DD	00273		PUSHL	R7	
	00000000G	00		03	FB	00275		CALLS	#3, DBG\$STA_SYMTYPE	
				7E	D4	0027C		CLRL	-(SP)	1173
			3C	AE	DD	0027E		PUSHL	TYPEID	
			44	AE	DD	00281		PUSHL	FCODE	
				57	DD	00284		PUSHL	R7	
			3C	AE	DD	00286		PUSHL	KIND	
				5B	DD	00289		PUSHL	PRIMPTR	
	C445	CF		06	FB	0028B		CALLS	#6, DBG\$BUILD_PRIMARY_SUBNODE	
		56	18	AB	DO	00290		MOVL	24(PRIMPTR), NODEPTR	1174
		13		59	E9	00294		BLBC	R9, 35\$	1174
		OF	04	AE	E9	00297		BLBC	4(SP), 35\$	
		07	3C	AE	D1	0029B		CMPL	FCODE, #7	1174
				05	12	0029F		BNEQ	34\$	
			10	A6	D4	002A1		CLRL	16(NODEPTR)	1175
				04	11	002A4		BRB	35\$	
	0A	A6		20	88	002A6	34\$:	BISB2	#32, 10(NODEPTR)	1175
		01	3C	AE	D1	002AA	35\$:	CMPL	FCODE, #1	1175
				03	13	002AE		BEQL	37\$	
				01BB	31	002B0	36\$:	BRW	61\$	
		F6	00000000	EF	E9	002B3	37\$:	BLBC	COMPONENTS IN PATHNAME, 36\$	1175
		52	28	A6	9E	002BA		MOVAB	40(R6), SUBVECTOR	1176
		50	1B	A6	9A	002BE		MOVZBL	27(NODEPTR), R0	1176
	20	AE		50	C0	002C2		ADDL2	R0, EXPECTED_SUBS	
	18	AE	1B	A6	9A	002C6		MOVZBL	27(NODEPTR), 24(SP)	1176
		59	08	AC	DO	002CB		MOVL	SUBSCR_DESC, R9	1177
		57		01	CE	002CF		MNEGL	#1, J	1187
				0158	31	002D2		BRW	58\$	
54	01	A9	08	00	ED	002D5	38\$:	CMPZV	#0, #8, 1(R9), SUBSCR_INDEX	1177
				10	14	002DB		BGTR	40\$	
			01	A9	95	002DD		TSTB	1(R9)	1177
				04	12	002E0		BNEQ	39\$	
				53	D5	002E2		TSTL	I	1177
				07	13	002E4		BEQL	40\$	
	1C	AE		01	DO	002E6	39\$:	MOVL	#1, TOOFEWSUB	1178
			01	A9	94	002EA		CLRB	1(R9)	1178
			01	A9	95	002ED	40\$:	TSTB	1(R9)	1179
				23	12	002F0		BNEQ	43\$	
		51	1B	A6	9A	002F2		MOVZBL	27(NODEPTR), R1	1179
		50		01	CE	002F6		MNEGL	#1, J	
				16	11	002F9		BRB	42\$	
	SA	54		50	C1	002FB	41\$:	ADDL3	J, SUBSCR_INDEX, R10	1179
		5A		0C	C4	002FF		MULL2	#12, R10	1179
			02	AA49	01	88	00302	BISB2	#1, 2(R10)[R9]	
				6A49	9F	00307		PUSHAB	(R10)[R9]	1179
		9E	00020000	8F	C8	0030A		BISL2	#131072, 2(SP)+	
		50		51	F3	00311	42\$:	AOBLEQ	R1, J, 41\$	1179
08	E6	57		14	C5	00315	43\$:	MULL3	#20, J, 8(SP)	1185
	AE		08	08	C1	0031A		ADDL3	#8, 8(SP), R0	1185
0C	50			50	C1	0031F		ADDL3	R0, SUBVECTOR, 12(SP)	
	AE	52		0C	C5	00324		MULL3	#12, SUBSCR_INDEX, R10	1180
	5A	54								

03	02	AA49	00	E0	00328	BBS	#0, 2(R10)[R9], 44\$...	
			0090	31	0032E	BRW	53\$...	
			53	D5	00331	TSTL	1	1181	
			1F	13	00333	BEQL	47\$...	
		01	A9	95	00335	TSTB	1(R9)	1181	
			1A	13	00338	BEQL	47\$...	
08	6A49		11	E1	0033A	BBC	#17, (R10)[R9], 45\$	1181	
		00028F68	8F	DD	0033F	PUSHL	#167784	1181	
			06	11	00345	BRB	46\$...	
		00028F60	8F	DD	00347	PUSHL	#167776	1182	
	00000000G	00	01	FB	0034D	CALLS	#1, LIB\$SIGNAL	...	
25	OA	A6	03	E0	00354	BBS	#3, 10(NODEPTR), 50\$	1182	
	OA	A6	08	88	00359	BISB2	#8, 10(NODEPTR)	1182	
		50	01	CE	0035D	MNEGL	#1, K	1182	
			18	11	00360	BRB	49\$...	
51	50		14	C5	00362	MULL3	#20, K, R1	...	
		08	A142	9F	00366	PUSHAB	8(R1)[SUBVECTOR]	1183	
			6142	9F	0036A	PUSHAB	(R1)[SUBVECTOR]	...	
	9E		9E	D0	0036D	MOVL	a(SP)+, a(SP)+	...	
		0C	A142	9F	00370	PUSHAB	12(R1)[SUBVECTOR]	1183	
			6142	9F	00374	PUSHAB	(R1)[SUBVECTOR]	...	
	9E		9E	D0	00377	MOVL	a(SP)+, a(SP)+	...	
E4	50		57	F2	0037A	AOBLSS	J, K, 48\$	1182	
34	6A49		11	E0	0037E	BBS	#17, (R10)[R9], 52\$	1183	
		08	AA49	9F	00383	PUSHAB	8(R10)[R9]	1184	
		04	AA49	9F	00387	PUSHAB	4(R10)[R9]	...	
	9E		9E	D1	0038B	CMPL	a(SP)+, a(SP)+	...	
			0D	15	0038E	BLEQ	51\$...	
		00028F08	8F	DD	00390	PUSHL	#167688	1184	
	00000000G	00	01	FB	00396	CALLS	#1, LIB\$SIGNAL	...	
50		57	14	C5	0039D	MULL3	#20, J, R0	1184	
			08	A042	9F	003A1	PUSHAB	8(R0)[SUBVECTOR]	1184
			04	AA49	9F	003A5	PUSHAB	4(R10)[R9]	...
	9E		9E	D0	003A9	MOVL	a(SP)+, a(SP)+	...	
		0C	A042	9F	003AC	PUSHAB	12(R0)[SUBVECTOR]	1185	
			08	AA49	9F	003B0	PUSHAB	8(R10)[R9]	...
	9E		9E	D0	003B4	MOVL	a(SP)+, a(SP)+	...	
		08	BE42	9F	003B7	PUSHAB	a8(SP)[SUBVECTOR]	1185	
	9E		10	BE	003BB	MOVL	a16(SP), a(SP)+	...	
			47	11	003BF	BRB	56\$	1185	
		04	AA49	9F	003C1	PUSHAB	4(R10)[R9]	1186	
			9E	D0	003C5	MOVL	a(SP)+, VAL	...	
	0C	BE	50	D1	003C8	CMPL	VAL, a12(SP)	1186	
			0F	19	003CC	BLSS	54\$...	
51	08	AE	0C	C1	003CE	ADDL3	#12, 8(SP), R1	1186	
6E		52	51	C1	003D3	ADDL3	R1, SUBVECTOR, (SP)	...	
	00	BE	50	D1	003D7	CMPL	VAL, a0(SP)	...	
			20	15	003DB	BLEQ	55\$...	
7E	08	AE	0C	C1	003DD	ADDL3	#12, 8(SP), -(SP)	1187	
			9E42	9F	003E2	PUSHAB	a(SP)+[SUBVECTOR]	...	
			9E	DD	003E5	PUSHL	a(SP)+	...	
		10	BE	DD	003E7	PUSHL	a16(SP)	1186	
			50	DD	003EA	PUSHL	VAL	1186	
			54	DD	003EC	PUSHL	SUBSCR_INDEX	...	
			04	DD	003EE	PUSHL	#4	...	
		00028688	8F	DD	003F0	PUSHL	#165515	...	
	00000000G	00	06	FB	003F6	CALLS	#6, LIB\$SIGNAL	...	

			08 BE42 9F 003FD 55\$:	PUSHAB	28(SP)[SUBVECTOR]	1187
			04 AA49 9F 00401	PUSHAB	4(R10)[R9]	
		9E	9E D0 00405	MOVL	2(SP)+, 2(SP)+	
1E	0A	A6	03 E1 00408 56\$:	BBC	#3, 10(NODEPTR), 57\$	1187
18	02	AA49	00 E0 0040D	BBS	#0, 2(R10)[R9], 57\$	1187
			08 BE42 9F 00413	PUSHAB	28(SP)[SUBVECTOR]	1188
		BE	9E D0 00417	MOVL	2(SP)+, 212(SP)	
50	0C	AE	0C C1 0041B	ADDL3	#12, 8(SP), R0	1188
51	08	52	50 C1 00420	ADDL3	R0, SUBVECTOR, R1	
		61	08 BE42 9F 00424	PUSHAB	28(SP)[SUBVECTOR]	
			9E D0 00428	MOVL	2(SP)+, (R1)	
		57	54 D6 0042B 57\$:	INCL	SUBSCR_INDEX	1188
02			18 AE F2 0042D 58\$:	AOBLSS	24(SP), J, 59\$	1176
			03 11 00432	BRB	60\$	
			FE9E 31 00434 59\$:	BRW	38\$	
	1F	A6	1B A6 90 00437 60\$:	MOVB	27(NODEPTR), 31(NODEPTR)	1189
2D	0A	A6	03 E0 0043C	BBS	#3, 10(NODEPTR), 61\$	1189
	0A	A6	01 88 00441	BISB2	#1, 10(NODEPTR)	1190
	38	AE	24 A6 D0 00445	MOVL	36(NODEPTR), TYPEID	1190
			38 AE DD 0044A	PUSHL	TYPEID	1190
	00000000G	00	01 FB 0044D	CALLS	#1, DBG\$STA_TYPEFCODE	
	3C	AE	50 D0 00454	MOVL	R0, FCODE	
			7E D4 00458	CLRL	-(SP)	1190
			3C AE DD 0045A	PUSHL	TYPEID	1190
		7E	44 AE DD 0045D	PUSHL	FCODE	
			06 7D 00460	MOVQ	#6, -(SP)	1190
			5B DD 00463	PUSHL	PRIMPTR	
	C26B	CF	06 FB 00465	CALLS	#6, DBG\$BUILD_PRIMARY_SUBNODE	
		56	18 AB D0 0046A	MOVL	24(PRIMPTR), NODEPTR	1190
		07	3C AE D1 0046E 61\$:	CMPL	FCODE, #7	1191
			42 12 00472	BNEQ	64\$	
			3B 00000000' EF E9 00474	BLBC	COMPONENTS_IN_PATHNAME, 64\$	1191
			53 D5 0047B	TSTL	J	1192
			37 13 0047D	BEQL	64\$	
	0A	A6	01 88 0047F	BISB2	#1, 10(NODEPTR)	1193
	14	AE	48 AE43 D0 00483	MOVL	SYMID_VECT-4[I], SYMID1	1193
		51	38 AE D0 00489	MOVL	TYPEID, R1	1193
		58	2C A1 9E 0048D	MOVAB	44(R1), TYPcomplst	
		50	01 CE 00491	MNEGL	#1, J	1194
			1B 11 00494	BRB	63\$	
			6840 D0 00496 62\$:	MOVL	(TYPcomplst)[J], SYMID2	1193
57	10	AE	0C C1 0049B	ADDL3	#12, SYMID2, R7	1194
59	10	AE	0C C1 004A0	ADDL3	#12, SYMID1, R9	
		67	69 D1 004A5	CMPL	(R9), (R7)	
			07 12 004A8	BNEQ	63\$	
18	A6	50	01 A1 004AA	ADDW3	#1, J, 24(NODEPTR)	1194
			05 11 004AF	BRB	64\$	1194
	E0	50	28 A1 F2 004B1 63\$:	AOBLSS	40(R1), J, 62\$	1193
		02	53 F4 004B6 64\$:	SOBGEQ	J, 65\$	1172
			03 11 004B9	BRB	66\$	
			FD84 31 004BB 65\$:	BRW	32\$	
			55 D4 004BE 66\$:	CLRL	PICKED_UP_SUBSTRING	1196
		57	08 AC D0 004C0	MOVL	SUBSCR_DESC, R7	1196
		50	01 A7 9A 004C4	MOVZBL	1(R7), -R0	
			50 D7 004C8	DECL	R0	
		50	54 D1 004CA	CMPL	SUBSCR_INDEX, R0	
			03 13 004CD	BEQL	68\$	

SA		54	0137	31	004CF	67\$:	BRW	80\$		
F3	02	AA47	0C	C5	004D2	68\$:	MULL3	#12, SUBSCR_INDEX, R10	1196	
SA		54	00	E1	004D6		BBC	#0, 2(R10)[R7], 67\$		
EA		6A47	0C	C5	004DC		MULL3	#12, SUBSCR_INDEX, R10	1196	
SA		54	12	E1	004E0		BBC	#18, (R10)[R7], 67\$		
E1		6A47	0C	C5	004E5		MULL3	#12, SUBSCR_INDEX, R10	1196	
			11	E0	004E9		BBS	#17, (R10)[R7], 67\$		
			44	AE	D4 004EE		CLRL	DTYPE	1196	
		06	2C	AE	D1 004F1		CMPL	KIND, #6	1197	
			78	12	004F5		BNEQ	73\$		
		50	3C	AE	D0 004F7		MOVL	FCODE, R0	1197	
		02	50	D1	004FB		CMPL	R0, #2	1197	
			17	12	004FE		BNEQ	69\$		
			40	AE	9F 00500		PUSHAB	BITSIZE	1197	
			48	AE	9F 00503		PUSHAB	DTYPE		
			40	AE	DD 00506		PUSHL	TYPEID		
52	00000000G	00	03	FB	00509		CALLS	#3, DBG\$STA_TYP_ATOMIC		
	40	AE	08	C7	00510		DIVL3	#8, BITSIZE, LEN	1197	
			33	11	00515		BRB	72\$	1197	
		03	50	D1	00517	69\$:	CMPL	R0, #3	1198	
			1B	12	0051A		BNEQ	70\$		
			48	AE	9F 0051C		PUSHAB	DESCR	1198	
			3C	AE	DD 0051F		PUSHL	TYPEID		
	00000000G	00	02	FB	00522		CALLS	#2, DBG\$STA_TYP_DESCR		
		50	48	AE	D0 00529		MOVL	DESCR, R0	1198	
	44	AE	02	A0	9A 0052D		MOVZBL	2(R0), DTYPE		
		52	60	3C	00532		MOVZWL	(R0), LEN	1198	
			13	11	00535		BRB	72\$	1197	
		05	50	D1	00537	70\$:	CMPL	R0, #5	1198	
			05	13	0053A		BEQL	71\$		
		07	50	D1	0053C		CMPL	R0, #7		
			09	12	0053F		BNEQ	72\$		
	44	AE	0E	D0	00541	71\$:	MOVL	#14, DTYPE	1198	
		52	8F	3C	00545		MOVZWL	#10000, LEN	1198	
		50	44	AE	D0 0054A	72\$:	MOVL	DTYPE, R0	1199	
		0E	50	D1	0054E		CMPL	R0, #14		
			21	13	00551		BEQL	74\$		
		0F	50	D1	00553		CMPL	R0, #15	1199	
			1C	13	00556		BEQL	74\$		
		10	50	D1	00558		CMPL	R0, #16	1199	
			17	13	0055B		BEQL	74\$		
		11	50	D1	0055D		CMPL	R0, #17	1199	
			12	13	00560		BEQL	74\$		
		12	50	D1	00562		CMPL	R0, #18	1199	
			0D	13	00565		BEQL	74\$		
		13	50	D1	00567		CMPL	R0, #19	1199	
			08	13	0056A		BEQL	74\$		
		14	50	D1	0056C		CMPL	R0, #20	1199	
			03	13	0056F	73\$:	BEQL	74\$		
			0088	31	00571		BRW	79\$		
	04	AB	01	8A	00574	74\$:	BICB2	#1, 4(PRIMPTR)	1200	
	04	AB	02	88	00578		BISB2	#2, 4(PRIMPTR)	1200	
SA		54	0C	C5	0057C		MULL3	#12, SUBSCR_INDEX, R10	1200	
			04	AA47	9F 00580		PUSHAB	4(R10)[R7]		
		53	9E	D0	00584		MOVL	@(SP)+, R3		
	0007FFFF	8F	53	D1	00587		CMPL	R3, #524287		
			11	15	0058E		BLEQ	75\$		

			53	DD	00590	PUSHL	R3	1201	
			01	DD	00592	PUSHL	#1	1201	
		000280F0	8F	DD	00594	PUSHL	#164080		
00000000G	00		03	FB	0059A	CALLS	#3, LIB\$SIGNAL		
10	AB		53	B0	005A1	75%: MOVW	R3, 16(PRIMPTR)	1201	
		08	AA47	9F	005A5	PUSHAB	8(R10)(R7)	1201	
	5A		9E	D0	005A9	MOVL	2(SP)+, R10		
	50		FF	AA43	9E	005AC	MOVAB	-1(R10)(R3), R0	1201
	52		50	D1	005B1	CMPL	R0, LEN	1201	
			04	14	005B4	BGTR	76%		
			53	D5	005B6	TSTL	R3	1201	
			15	14	005B8	BGTR	77%		
			52	DD	005BA	76%: PUSHL	LEN	1202	
		0408	8F	BB	005BC	PUSHR	#^M<R3,R10>	1202	
			03	DD	005C0	PUSHL	#3	1202	
		000280D8	8F	DD	005C2	PUSHL	#164056		
00000000G	00		05	FB	005C8	CALLS	#5, LIB\$SIGNAL		
00007FFF	8F		5A	D1	005CF	77%: CMPL	R10, #32767	1202	
			11	15	005D6	BLEQ	78%		
			5A	DD	005D8	PUSHL	R10	1203	
			01	DD	005DA	PUSHL	#1	1202	
		000280F8	8F	DD	005DC	PUSHL	#164088		
00000000G	00		03	FB	005E2	CALLS	#3, LIB\$SIGNAL		
12	AB		5A	B0	005E9	78%: MOVW	R10, 18(PRIMPTR)	1203	
	56	18	AB	D0	005ED	MOVL	24(PRIMPTR), NODEPTR	1203	
14	A6		01	CE	005F1	MNEGL	#1, 20(NODEPTR)	1203	
	55		01	D0	005F5	MOVL	#1, PICKED_UP_SUBSTRING	1203	
			54	D6	005F8	INCL	SUBSCR_INDEX	1203	
			DD	11	005FA	BRB	80%	1199	
		00028F70	8F	DD	005FC	79%: PUSHL	#167792	1205	
00000000G	00		01	FB	00602	CALLS	#1, LIB\$SIGNAL		
	12	1C	AE	E9	00609	80%: BLBC	TOOFEWSUB, 81%	1205	
		20	AE	DD	0060D	PUSHL	EXPECTED_SUBS	1206	
			01	DD	00610	PUSHL	#1		
		00028EA0	8F	DD	00612	PUSHL	#167584		
			03	FB	00618	CALLS	#3, LIB\$SIGNAL		
54	01	A7	00	ED	0061F	81%: CMPZV	#0, #8, 1(R7), SUBSCR_INDEX	1206	
			23	15	00625	BLEQ	84%		
	7E	54	55	C3	00627	SUBL3	PICKED_UP_SUBSTRING, SUBSCR_INDEX, -(SP)	1206	
			01	DD	0062B	PUSHL	#1		
		00028EB0	8F	DD	0062D	PUSHL	#167600		
			0E	11	00633	BRB	83%		
		00000000'	EF	9F	00635	82%: PUSHAB	P.AYS	1207	
			01	DD	0063B	PUSHL	#1		
		00028362	8F	DD	0063D	PUSHL	#164706		
00000000G	00		03	FB	00643	83%: CALLS	#3, LIB\$SIGNAL		
	50		5B	D0	0064A	84%: MOVL	PRIMPTR, R0	1207	
			04	0064D	RET			1207	

; Routine Size: 1614 bytes, Routine Base: DBG\$CODE + 3997

```
:11992 12079 1
:11993 12080 1
:11994 12081 1
:11995 12082 1
:11996 12083 1
:11997 12084 1
:11998 12085 1
:11999 12086 1
:12000 12087 1
:12001 12088 1
:12002 12089 1
:12003 12090 1
:12004 12091 1
:12005 12092 1
:12006 12093 1
:12007 12094 1
:12008 12095 1
:12009 12096 1
:12010 12097 1
:12011 12098 1
:12012 12099 1
:12013 12100 1
:12014 12101 1
:12015 12102 1
:12016 12103 1
:12017 12104 1
:12018 12105 1
:12019 12106 1
:12020 12107 1
:12021 12108 1
:12022 12109 1
:12023 12110 1
:12024 12111 1
:12025 12112 1
:12026 12113 1
:12027 12114 1
:12028 12115 2
:12029 12116 2
:12030 12117 2
:12031 12118 2
:12032 12119 2
:12033 12120 2
:12034 12121 2
:12035 12122 2
:12036 12123 2
:12037 12124 2
:12038 12125 2
:12039 12126 2
:12040 12127 2
:12041 12128 2
:12042 12129 2
:12043 12130 2
:12044 12131 2
:12045 12132 2
:12046 12133 2
:12047 12134 2
:12048 12135 2
```

ROUTINE RESOLVE_COMPONENT (TYPEID, COMP_LIST, SYMID, PRIMPTR, COMPNAME) =

FUNCTION

This routine is called from GET_RECORD_COMPONENT to resolve possible ambiguities where the user has specified X.Y and there is more than one record component named Y.

This situation arises in C, where membership checking is not enforced. Thus in C you can say X.Y even though Y is not a component in the record given by X. If we find more than one symid for Y, there is a possible ambiguity (although if all of the symids are record components with the same type and offset, then it is OK).

A different situation arises in BASIC. Here, we allow A::C to be an abbreviation for A::B::C (incomplete data qualification). In this case, the INCOMPLETE_QUAL flag is lit. What we do here is call a routine which chases upscope pointers to determine whether we can get to the given TYPEID from the component SYMID.

INPUTS

TYPEID - typeid for the record
COMP_LIST - list of symids which are record components having the same name. This list is in the form of a vector of longword, with the first longword being the count.
SYMID - address in which to leave the resolved symid, if one is determined.
PRIMPTR - pointer to the input Primary
COMPNAME - name of the component

OUTPUTS

The value TRUE is returned if a unique symid was determined; FALSE otherwise. If TRUE is returned then the output parameter SYMID is filled in.

BEGIN

MAP

COMP_LIST: REF VECTOR[];

LOCAL

FOUND;

DBG\$GL_CURRENT_PRIMARY = .PRIMPTR;

! If incomplete data qualification is allowed in this language (for example A::C in place of A::B::C) then we search the list of candidate components to see if there is a unique one in the given record. For example, if the user says A::C, and there are several components C, but only one belongs in record A, then that is the one we want.

This is the code path taken for language BASIC.

IF .INCOMPLETE_QUAL

THEN

BEGIN

FOUND = FALSE;

```
:12049      12136      INCR I FROM 1 TO .COMP_LIST[0] DO
:12050      12137
:12051      12138      | The CHECK_UPSCOPE routine determines whether the given
:12052      12139      | record component belongs in the record given by TYPEID.
:12053      12140      | If it does, but there are intervening record components,
:12054      12141      | then the Primary must be modified to include the component
:12055      12142      | selection for these intervening components.
:12056      12143      IF CHECK_UPSCOPE(.COMP_LIST[I], .TYPEID, .PRIMPTR, 0)
:12057      12144      THEN
:12058      12145          IF .FOUND
:12059      12146          THEN
:12060      12147              | Not unique.
:12061      12148              RETURN FALSE
:12062      12149          ELSE
:12063      12150              BEGIN
:12064      12151                  | We found one.
:12065      12152                  .SYMID = .COMP_LIST[I];
:12066      12153                  FOUND = TRUE;
:12067      12154                  END;
:12068      12155
:12069      12156      | If we failed to find one, signal that the given component
:12070      12157      | is not a field of this record. Note that we do not return FALSE
:12071      12158      | here - returning FALSE indicates an ambiguous field name.
:12072      12159      IF NOT .FOUND
:12073      12160      THEN
:12074      12161          SIGNAL(DBG$_NOFIELD, 1, .COMPNAME);
:12075      12162
:12076      12163      | Found a unique component - return true.
:12077      12164      RETURN TRUE;
:12078      12165      END;
:12079      12166
:12080      12167      | This is where we end up for C: we are trying to resolve an ambiguity.
:12081      12168      | Check for only one component in list: then there is no ambiguity.
:12082      12169      IF .COMP_LIST[0] EQL 1
:12083      12170      THEN
:12084      12171          BEGIN
:12085      12172              .SYMID = .COMP_LIST[1];
:12086      12173              RETURN TRUE;
:12087      12174          END;
:12088      12175
:12089      12176      | More than one component.
:12090      12177      RETURN FALSE;
:12091      12178      END;
:12092      12179
:12093      12180
:12094      12181
:12095      12182
:12096      12183
:12097      12184
:12098      12185
:12099      12186
:12100      12187
:12101      12188
```


000C 00000 RESOLVE_COMPONENT:							
00000000G	00	10	AC	D0	00002	.WORD Save R2,R3	1207
	40	00000000	EF	E9	0000A	MOVL PRIMPTR,DBG\$GL_CURRENT_PRIMARY	1212
			52	7C	00011	BLBC INCOMPLETE_QUAL,3\$	1213
			20	11	00013	CLRQ 1	1214
			7E	D4	00015	BRB 2\$	
			10	AC	DD 00017	CLRL -(SP)	
		04	AC	DD 0001A		PUSHL PRIMPTR	
		08	BC42	DD 0001D		PUSHL TYPEID	
E2FB	CF		04	FB	00021	PUSHL @COMP_LIST[1]	
	OC		50	E9	00026	CALLS #4,CHECK_UPSCOPE	
	37		53	E8	00029	BLBC R0,2\$	1214
OC	BC	08	BC42	D0	0002C	BLBS FOUND,5\$	1215
	53		01	D0	00032	MOVL @COMP_LIST[1],@SYMID	1215
DB	52	08	BC	F3	00035	MOVL #1,FOUND	1214
	22		53	E8	0003A	AOBLEQ @COMP_LIST,1,1\$	1216
		14	AC	DD 0003D		BLBS FOUND,4\$	1216
			01	DD 00040		PUSHL COMPNAME	
			8F	DD 00042		PUSHL #1	
00000000G	00	00028C80	03	FB	00048	PUSHL #167040	
			0E	11	0004F	CALLS #3,LIB\$SIGNAL	
	50	08	AC	D0	00051	BRB 4\$	1217
	01		60	D1	00055	MOVL COMP_LIST,R0	1217
			09	12	00058	CML (R0),#1	
OC	BC	04	A0	D0	0005A	BNEQ 5\$	1218
	50		01	D0	0005F	MOVL 4(R0),@SYMID	1218
				04	00062	MOVL #1,R0	
			50	D4	00063	RET	1218
			04	00065		CLRL R0	
						RET	

; Routine Size: 102 bytes, Routine Base: DBG\$CODE + 3FE5

```
:12103
:12104
:12105
:12106
:12107
:12108
:12109
:12110
:12111
:12112
:12113
:12114
:12115
:12116
:12117
:12118
:12119
:12120
:12121
:12122
:12123
:12124
:12125
:12126
:12127
:12128
:12129
:12130
:12131
:12132
:12133
:12134
:12135
:12136
:12137
:12138
:12139
:12140
:12141
:12142
:12143
:12144
:12145
:12146
:12147
:12148
:12149
:12150
:12151
:12152
:12153
:12154
:12155
:12156
:12157
:12158
:12159
```

```
12189
12190
12191
12192
12193
12194
12195
12196
12197
12198
12199
12200
12201
12202
12203
12204
12205
12206
12207
12208
12209
12210
12211
12212
12213
12214
12215
12216
12217
12218
12219
12220
12221
12222
12223
12224
12225
12226
12227
12228
12229
12230
12231
12232
12233
12234
12235
12236
12237
12238
12239
12240
12241
12242
12243
12244
12245
```

```
ROUTINE SAVE_SUBSCRIPTS(PATHDESC, SUBSCR_DESC): NOVALUE =
```

```
FUNCTION
```

```
When parsing PLI or BASIC Primaries we want to save away the subscripts
that we see along the way. This routine picks up subscripts and
saves them into the SUBSCR_DESC data structure.
```

```
INPUTS
```

```
PATHDESC      -      A pointer to the pathname descriptor for
                      the pathname we have parsed so far.
SUBSCR_DESC    -      A data structure containing the subscript
                      values.
```

```
OUTPUTS
```

```
BEGIN
```

```
MAP
```

```
PATHDESC: REF PTH$PATHNAME,
SUBSCR_DESC: REF SUBSCR$DESC;
```

```
LOCAL
```

```
DECLTYPE: REF DBG$VALDESC,      | Pointer to Value Descriptor for
                                | declared subscript data type
LA_PTR: REF VECTOR[.BYTE],      | Lookahead pointer into input
LOW_RANGE_VAL,                  | Low value of a subscript range
PATH_INDEX,                     | Count of pathname components
SAVED_RADIX,                    | Temporarily saved expression radix
SUBSCR_COUNT,                   | Actual subscript count in input line
THIS_SUBSCR_IS_RANGE,           | Flag set if the current subscript is
                                | given as a subscript range
TOKEN,                          | Lexical Token
VALADDR: REF VECTOR[.LONG],     | Pointer to integer subscript value
VALPTR: REF DBG$VALDESC;        | Pointer to subscript Value Descriptor
```

```
! Note how many subscripts we have previously picked up (i.e., in earlier
! calls to this routine, while parsing this same expression). In PL/I,
! for example, the subscripts may arrive in separate pieces, and do not
! necessarily have to be associated with the "right" component, e.g.,
! X(1,2).Y(3).Z(4,5)
```

```
! This routine also picks up substring references in COBOL. Because of this,
! we set a marker to indicate where we were in SUBSCR_DESC when we entered
! this routine. E.g., in the COBOL expression X(1,2,3)(1:5) we set a
! marker to indicate that the 1:5 came in a separate set of parenthesis.
! It would be illegal otherwise, so this marker can be used later to
! decide whether to signal an error.
```

```
SUBSCR_COUNT = .SUBSCR_DESC[0, SUBSCR$B SUBCNT];
SUBSCR_DESC[.SUBSCR_COUNT, SUBSCR$V_MARKER] = TRUE;
```

```
! Loop through the subscript expressions for this array reference. Each
! subscript is parsed, evaluated, and converted to integer. Its value
! is then stored in the SUBSCR_DESC data structure.
```

```
PATH_INDEX = .PATHDESC[PTH$B_TOTCNT];
```

```
:12160
:12161
:12162
:12163
:12164
:12165
:12166
:12167
:12168
:12169
:12170
:12171
:12172
:12173
:12174
:12175
:12176
:12177
:12178
:12179
:12180
:12181
:12182
:12183
:12184
:12185
:12186
:12187
:12188
:12189
:12190
:12191
:12192
:12193
:12194
:12195
:12196
:12197
:12198
:12199
:12200
:12201
:12202
:12203
:12204
:12205
:12206
:12207
:12208
:12209
:12210
:12211
:12212
:12213
:12214
:12215
:12216
```

```
12246
12247
12248
12249
12250
12251
12252
12253
12254
12255
12256
12257
12258
12259
12260
12261
12262
12263
12264
12265
12266
12267
12268
12269
12270
12271
12272
12273
12274
12275
12276
12277
12278
12279
12280
12281
12282
12283
12284
12285
12286
12287
12288
12289
12290
12291
12292
12293
12294
12295
12296
12297
12298
12299
12300
12301
12302
```

```
THIS SUBSCR IS RANGE = FALSE;
TERMINATOR_CODE = TOKEN$K_TERM_COMMA;
WHILE TERMINATOR_CODE NEQ TOKEN$K_TERM_CLOSE DO
  BEGIN

    ! Look for the asterisk. X(*) is the same as X(lower:upper).
    ! If we find the asterisk then advance the character pointer beyond
    ! the asterisk and also increment the subscript count.

    LA_PTR = .CHARPTR;
    WHILE .LA_PTR[0] EQL ' ' DO LA_PTR = .LA_PTR + 1;
    IF .LA_PTR[0] EQL '*'
    THEN
      BEGIN
        CHARPTR = .LA_PTR + 1;

        ! Call the Lexical Scanner to take us past the ',', ' or
        ! or ']' or ')'. This will set TERMINATOR_CODE to the
        ! terminator that is seen. If we do not see a terminator
        ! then signal a syntax error. Also signal an error if
        ! ':' was the terminator.

        TOKEN = DBG$LEXICAL_SCANNER (FALSE, FALSE,
          SUBSCRIPT_TERM_TBL, 0);
        IF .TOKEN NEQ TERMINATOR_TOKEN
        THEN
          BEGIN
            LOCAL
              ASCII_STRING: VECTOR[2,BYTE];
              ASCII_STRING[0] = 1;
              ASCII_STRING[1] = .CHARPTR[0];
              SIGNAL(DBG$SYNERREXPR, 1, ASCII_STRING);
            END;
          IF .TERMINATOR_CODE EQL TOKEN$K_TERM_COLON
          THEN
            SIGNAL(DBG$INVRANSPEC);
          IF .TERMINATOR_CODE EQL TOKEN$K_TERM_NONE
          THEN
            SIGNAL(DBG$MISCLOSUB);
          CHARPTR = .CHARPTR + .TERMINATOR_LENGTH;

          SUBSCR_DESC[.SUBSCR_COUNT, SUBSCR$B_PATH_INDEX] = .PATH_INDEX;
          SUBSCR_DESC[.SUBSCR_COUNT, SUBSCR$V_ASTER] = TRUE;
          SUBSCR_DESC[.SUBSCR_COUNT, SUBSCR$V_RANGE] = TRUE;
          SUBSCR_COUNT = .SUBSCR_COUNT + 1;
        END
      ELSE
        BEGIN

          ! Call the expression parser to pick up the next subscript expression
          ! and its value. Note that we set the radix to decimal over this call
          ! and then restore it. Also note that the Expression Parser sets
```

```

:12217 12303 4
:12218 12304 4
:12219 12305 4
:12220 12306 4
:12221 12307 4
:12222 12308 4
:12223 12309 4
:12224 12310 4
:12225 12311 4
:12226 12312 4
:12227 12313 4
:12228 12314 4
:12229 12315 4
:12230 12316 4
:12231 12317 4
:12232 12318 4
:12233 12319 4
:12234 12320 4
:12235 12321 4
:12236 12322 4
:12237 12323 4
:12238 12324 4
:12239 12325 4
:12240 12326 4
:12241 12327 4
:12242 12328 4
:12243 12329 4
:12244 12330 4
:12245 12331 4
:12246 12332 4
:12247 12333 4
:12248 12334 4
:12249 12335 4
:12250 12336 4
:12251 12337 4
:12252 12338 4
:12253 12339 4
:12254 12340 4
:12255 12341 4
:12256 12342 4
:12257 12343 4
:12258 12344 4
:12259 12345 4
:12260 12346 4
:12261 12347 4
:12262 12348 4
:12263 12349 4
:12264 12350 4
:12265 12351 5
:12266 12352 5
:12267 12353 5
:12268 12354 5
:12269 12355 5
:12270 12356 5
:12271 12357 5
:12272 12358 5
:12273 12359 5

```

```

! TERMINATOR_CODE and TERMINATOR_LENGTH as a side-effect.
!
! SAVED RADIX = .EXPRESSION RADIX;
! EXPRESSION RADIX = DBG$K_DECIMAL;
! VALPTR = DBG$EXPRESSION_PARSER (FALSE, .SUBSCRIPT_TERM_TBL);
! EXPRESSION_RADIX = .SAVED_RADIX;
!
! Check the terminator code. If there was no terminator (the input
! line just ended), signal an error. Otherwise we got a comma or clos-
! ing subscript parenthesis and we increment CHARPTR to get past it.
!
! IF .TERMINATOR_CODE EQL TOKEN$K_TERM_NONE THEN SIGNAL(DBG$_MISCLOSUB);
! CHARPTR = .CHARPTR + .TERMINATOR_LENGTH;
!
! We now need to convert the subscript to one of the appropriate
! dtype. We need to set up a target descriptor for the conversion
! routine. We allocate a skeleton descriptor and fill in some of
! the fields.
!
! DECLTYPE = DBG$MAKE_SKELETON_DESC(DBG$K_VALUE_DESC, 4);
! DECLTYPE[DBG$B_DHDR_KIND] = RST$K_DATA;
! DECLTYPE[DBG$B_DHDR_FCODE] = RST$K_TYPE_ATOMIC;
! DECLTYPE[DBG$B_VALUE_CLASS] = DSC$K_CLASS_S;
! DECLTYPE[DBG$B_VALUE_DTYPE] = DSC$K_DTYPE_L;
! DECLTYPE[DBG$W_VALUE_LENGTH] = 4;
! DECLTYPE[DBG$L_VALUE_POINTER] = DECLTYPE[DBG$A_VALUE_ADDRESS];
!
! Finally call the conversion routine. This routine checks that
! the conversion is legal before doing it.
!
! VALPTR = DBG$EVAL_LANG_OPERATOR(DBG$GL_CONVERT_TOKEN, .VALPTR, .DECLTYPE);
! VALADDR = .VALPTR[DBG$C_VALUE_POINTER];
!
! If the terminator at the end of this subscript expression was a colon
! we have a subscript range (for example, "ARR(1:5,2)"). We thus set
! the subscript-range flag and save the low value of the range, i.e.
! the value we just picked up. If this is the first range in the
! subscript list, we also turn all previous subscripts into ranges by
! setting the lower and upper bound for each such subscript to the
! corresponding subscript value. This in effect defines a new array
! which constitutes a "slice" of the original array.
!
! IF .TERMINATOR_CODE EQL TOKEN$K_TERM_COLON
! THEN
!   BEGIN
!     IF .THIS_SUBSCR_IS_RANGE THEN SIGNAL(DBG$_INVTRANSPEC);
!     THIS_SUBSCR_IS_RANGE = TRUE;
!     LOW_RANGE_VAL = .VALADDR[0];
!   END
!
! The terminator was not a colon, so we now have the full subscript
! specification. Fill the subscript value into the Array Sub-Node's

```



```
:12274      12360      3      ! subscript vector. Set up the bounds for an array 'slice' if this
:12275      12361      3      ! or any previous subscript specification in this array reference
:12276      12362      3      ! consisted of a subscript range. Also bump the subscript count.
:12277      12363      3      ELSE
:12278      12364      4      BEGIN
:12279      12365      5
:12280      12366      5
:12281      12367      5
:12282      12368      5      ! If this subscript is specified as a subscript range, check that
:12283      12369      5      ! the first value in the range is not greater than the second.
:12284      12370      5      ! Also clear the subscript-is-range flag for the next subscript.
:12285      12371      5
:12286      12372      5      IF .THIS_SUBSCR_IS_RANGE
:12287      12373      5      THEN
:12288      12374      6          BEGIN
:12289      12375      6              SUBSCR_DESC[.SUBSCR_COUNT, SUBSCR$V_RANGE] = TRUE;
:12290      12376      6              THIS_SUBSCR_IS_RANGE = FALSE;
:12291      12377      6              END
:12292      12378      6
:12293      12379      6
:12294      12380      6      ! Otherwise, set the low range value to be the subscript value.
:12295      12381      6
:12296      12382      5      ELSE
:12297      12383      5          LOW_RANGE_VAL = .VALADDR[0];
:12298      12384      5
:12299      12385      5
:12300      12386      5      ! Finally fill in the subscript value itself (the start of the
:12301      12387      5      ! range). Increment the subscript count, and loop.
:12302      12388      5
:12303      12389      5      SUBSCR_DESC[.SUBSCR_COUNT, SUBSCR$B_PATH_INDEX] = .PATH_INDEX;
:12304      12390      5      SUBSCR_DESC[.SUBSCR_COUNT, SUBSCR$L_LBOUND] = .LOW_RANGE_VAL;
:12305      12391      5      IF .SUBSCR_DESC[.SUBSCR_COUNT, SUBSCR$V_RANGE]
:12306      12392      5      THEN
:12307      12393      5          SUBSCR_DESC[.SUBSCR_COUNT, SUBSCR$L_UBOUND] = .VALADDR[0];
:12308      12394      5          SUBSCR_COUNT = .SUBSCR_COUNT + 1;
:12309      12395      4      END;
:12310      12396      3      END;
:12311      12397      2      END;      ! End of WHILE loop over subscripts
:12312      12398      2
:12313      12399      2
:12314      12400      2      ! We have picked up all the subscripts within this set of subscript paren-
:12315      12401      2      ! theses.
:12316      12402      2      SUBSCR_DESC[0, SUBSCR$B_SUBCNT] = .SUBSCR_COUNT;
:12317      12403      2      RETURN;
:12318      12404      1      END;
```

```
50      OFFC 00000 SAVE_SUBSCRIPTS:
      SE      08      C2 00002      .WORD      Save R2,R3,R4,R5,R6,R7,R8,R9,R10,R11      : 1218
      53      08      AC D0 00005      SUBL2      #8, SP
      54      01      A3 9A 00009      MOVL      SUBSCR_DESC, R3      : 1223
      54      0C      C5 0000D      MOVZBL     1(R3), SUBSCR_COUNT
      6043 9F 00011      MULL3      #12, SUBSCR_COUNT, R0      : 1223
      PUSHAB (R0)[R3]
```

	9E 00040000	8F C8 00014	BISL2	#262144, @ (SP)+	
	59 04	BC 9A 0001B	MOVZBL	@PATHDESC, PATH_INDEX	1224
		58 D4 0001F	CLRL	THIS SUBSCR IS RANGE	1224
00000000'	EF 02 00000000'	01 D0 00021	MOVL	#1, TERMINATOR_CODE	1224
		EF D1 00028	1%: CMPL	TERMINATOR_CODE, #2	1224
		03 12 0002F	BNEQ	2%	
		01 81 31 00031	BRW	17%	
	56 00000000'	EF D0 00034	2%: MOVL	CHARPTR, LA_PTR	1225
	20	66 91 0003B	3%: CMPB	(LA_PTR), #32	1225
		04 12 0003E	BNEQ	4%	
		56 D6 00040	INCL	LA_PTR	
		F7 11 00042	BRB	3%	
	2A	66 91 00044	4%: CMPB	(LA_PTR), #42	1225
		03 13 00047	BEQL	5%	
		00 94 31 00049	BRW	9%	
00000000'	EF 01	A6 9E 0004C	5%: MOVAB	1(R6), CHARPTR	1226
		7E D4 00054	CLRL	-(SP)	1227
	00000000'	EF DD 00056	PUSHL	SUBSCRIPT_TERM_TBL	1227
		7E 7C 0005C	CLRL	-(SP)	1227
C9F0	CF	04 FB 0005E	CALLS	#4, DBG\$LEXICAL_SCANNER	
	6E	50 D0 00063	MOVL	R0, TOKEN	
	50 00000000'	EF 9E 00066	MOVAB	TERMINATOR_TOKEN, R0	1227
	50	6E D1 0006D	CMPL	TOKEN, R0	
		1E 13 00070	BEQL	6%	
	04 AE	01 90 00072	MOVB	#1, ASCII_STRING	1227
	05 AE 00000000'	FF 90 00076	MOVB	@CHARPTR, ASCII_STRING+1	1227
		04 AE 9F 0007E	PUSHAB	ASCII_STRING	1227
		01 DD 00081	PUSHL	#1	
	000289E2	8F DD 00083	PUSHL	#166370	
00000000G	00	03 FB 00089	CALLS	#3, LIB\$SIGNAL	
	03 00000000'	EF D1 00090	6%: CMPL	TERMINATOR_CODE, #3	1228
		0D 12 00097	BNEQ	7%	
	00028F08	8F DD 00099	PUSHL	#167688	1228
00000000G	00	01 FB 0009F	CALLS	#1, LIB\$SIGNAL	
	00000000'	EF D5 000A6	7%: TSTL	TERMINATOR_CODE	1228
		0D 12 000AC	BNEQ	8%	
	00028E90	8F DD 000AE	PUSHL	#167568	1228
00000000G	00	01 FB 000B4	CALLS	#1, LIB\$SIGNAL	
50 00000000'	EF 00000000'	EF C0 000BB	8%: ADDL2	TERMINATOR_LENGTH, CHARPTR	1228
	54	0C C5 000C6	MULL3	#12, SUBSCR_COUNT, R0	1229
	6043	59 90 000CA	MOVB	PATH_INDEX, -(R0)[R3]	
		6043 9F 000CE	PUSHAB	(R0)[R3]	1229
	9E 00020000	8F C8 000D1	BISL2	#131072, @ (SP)+	
	02 A043	01 88 000D8	BISB2	#1, 2(R0)[R3]	1229
		00 D0 31 000DD	BRW	15%	1229
	5B 00000000'	EF D0 000E0	9%: MOVL	EXPRESSION_RADIX, SAVED_RADIX	1230
00000000'	EF	0A D0 000E7	MOVL	#10, EXPRESSION_RADIX	1230
	00000000'	EF DD 000EE	PUSHL	SUBSCRIPT_TERM_TBL	1230
		7E D4 000F4	CLRL	-(SP)	
	C4B2	CF	02 FB 000F6	CALLS	#2, DBG\$EXPRESSION_PARSER
	55	50 D0 000FB	MOVL	R0, VALPTR	
00000000'	EF	5B D0 000FE	MOVL	SAVED_RADIX, EXPRESSION_RADIX	1230
	00000000'	EF D5 00105	TSTL	TERMINATOR_CODE	1231
		0D 12 0010B	BNEQ	10%	
	00028E90	8F DD 0010D	PUSHL	#167568	
00000000G	00	01 FB 00113	CALLS	#1, LIB\$SIGNAL	
00000000'	EF 00000000'	EF C0 0011A	10%: ADDL2	TERMINATOR_LENGTH, CHARPTR	1231

			04	DD	00125	PUSHL	#4	1232
			8F	9A	00127	MOVZBL	#122, -(SP)	
00000000G	7E	7A	02	FB	00128	CALLS	#2, DBG\$MAKE_SKELETON_DESC	
	52		50	DD	00132	MOVL	R0, DECLTYPE	
	06	0602	8F	BD	00135	MOVW	#1538, 6(DECLTYPE)	1232
	14	01080004	8F	DD	00138	MOVL	#17301508, 20(DECLTYPE)	1232
	18	20	A2	9E	00143	MOVAB	32(R2), 24(DECLTYPE)	1233
			52	DD	00148	PUSHL	DECLTYPE	1233
			55	DD	0014A	PUSHL	VALPTR	
		00000000'	EF	9F	0014C	PUSHAB	DBG\$GL_CONVERT_TOKEN	
00000000G	00		03	FB	00152	CALLS	#3, DBG\$EVAL_LANG_OPERATOR	
	55		50	DD	00159	MOVL	R0, VALPTR	
	57	18	A5	DD	0015C	MOVL	24(VALPTR), VALADDR	1233
	03	00000000'	EF	D1	00160	CMPL	TERMINATOR_CODE, #3	1234
			18	12	00167	BNEQ	12\$	
	0D		58	E9	00169	BLBC	THIS_SUBSCR_IS_RANGE, 11\$	1235
		00028F08	8F	DD	0016C	PUSHL	#167888	
00000000G	00		01	FB	00172	CALLS	#1, LIB\$SIGNAL	
	58		01	DD	00179	MOVL	#1, THIS_SUBSCR_IS_RANGE	1235
	5A		67	DD	0017C	MOVL	(VALADDR), LOW_RANGE_VAL	1235
			31	11	0017F	BRB	16\$	1234
	0D		58	E9	00181	BLBC	THIS_SUBSCR_IS_RANGE, 13\$	1237
50	54		0C	C5	00184	MULL3	#12, SUBSCR_COUNT, R0	1237
	02	A043	01	88	00188	BISB2	#1, 2(R0)[R3]	
			58	D4	0018D	CLRL	THIS_SUBSCR_IS_RANGE	1237
			03	11	0018F	BRB	14\$	1237
	5A		67	DD	00191	MOVL	(VALADDR), LOW_RANGE_VAL	1238
50	54		0C	C5	00194	MULL3	#12, SUBSCR_COUNT, R0	1238
	6043		59	90	00198	MOVB	PATH_INDEX, (R0)[R3]	
		04	A043	9F	0019C	PUSHAB	4(R0)[R3]	1239
	9E		5A	DD	001A0	MOVL	LOW_RANGE_VAL, a(SP)+	
07	02	A043	00	E1	001A3	BBC	#0, 2(R0)[R3], 15\$	1239
		08	A043	9F	001A9	PUSHAB	8(R0)[R3]	1239
	9E		67	DD	001AD	MOVL	(VALADDR), a(SP)+	
			54	D6	001B0	INCL	SUBSCR_COUNT	1239
		FE73	31	001B2	16\$:	BRW	1\$	1224
	01	A3	54	90	001B5	MOVB	SUBSCR_COUNT, 1(R3)	1240
			04	001B9	17\$:	RET		1240

; Routine Size: 442 bytes, Routine Base: DBG\$CODE + 404B

```
:12320 12405 1
:12321 12406 1
:12322 12407 1
:12323 12408 1
:12324 12409 1
:12325 12410 1
:12326 12411 1
:12327 12412 1
:12328 12413 1
:12329 12414 1
:12330 12415 1
:12331 12416 1
:12332 12417 1
:12333 12418 1
:12334 12419 1
:12335 12420 1
:12336 12421 1
:12337 12422 1
:12338 12423 1
:12339 12424 1
:12340 12425 1
:12341 12426 1
:12342 12427 1
:12343 12428 1
:12344 12429 1
:12345 12430 1
:12346 12431 1
:12347 12432 1
:12348 12433 1
:12349 12434 1
:12350 12435 1
:12351 12436 1
:12352 12437 1
:12353 12438 1
:12354 12439 1
:12355 12440 1
:12356 12441 1
:12357 12442 1
:12358 12443 1
:12359 12444 2
:12360 12445 2
:12361 12446 2
:12362 12447 2
:12363 12448 2
:12364 12449 2
:12365 12450 2
:12366 12451 2
:12367 12452 2
:12368 12453 2
:12369 12454 2
:12370 12455 2
:12371 12456 2
:12372 12457 2
:12373 12458 2
:12374 12459 2
:12375 12460 2
:12376 12461 2
```

ROUTINE SCAN_QUOTED_STRING(TOKENBUFFER, TOKEN_TYPE): NOVALUE =

FUNCTION

This routine scans a quoted character string and returns the found string in Counted ASCII format to a caller-provided buffer. It expects the OWN pointer CHARPTR to point to the quote character at the start of the character constant and it assumes that this in fact is a valid quote character. It then scans for the closing quote character (which must be the same character as the opening quote), treating doubled up quote characters within the string as a single quoted quote character. If the language is set to PL/I, then after finding the closing quote, it searches for a 'B' or 'b', to determine if the string could be a bit-string rather than a character string. If the quote characters are not '"', or if any double quotes are encountered, then the trailing 'B' is not searched for.

If a carriage-return (end of input line) is found before the closing quote or if the string exceeds 255 characters (the longest Counted ASCII allows), an error is signalled. Otherwise, CHARPTR is left pointing to the first character after the closing quote and the string itself is returned as Counted ASCII to the caller's buffer.

INPUTS

TOKENBUFFER - A pointer to the buffer in which the character string is to be accumulated. This buffer is expected to be 256 characters long, enough for the longest Counted ASCII string.

TOKEN_TYPE - Address of where to return token's type.

OUTPUTS

TOKENBUFFER - The quoted character string is accumulated and returned as a Counted ASCII string to the buffer pointed to by TOKENBUFFER.

TOKEN_TYPE - Is filled in with either TOKEN\$K_STRING or TOKEN\$K_BIT_STRING.

BEGIN

MAP

TOKENBUFFER: REF VECTOR[BYTE], ! Pointer to buffer for char string
TOKEN_TYPE : REF VECTOR[1]; ! Longword to receive token's type.

LOCAL

QUOTE, ! Quote character which started the
! current quoted string constant
DOUBLED_QUOTES, ! flag denotes whether or not there
! exist doubled up quotes ('').
TOKENLEN; ! Length of quoted string so far

! We pick up the closing quote character (which must be the same as the
! opening quote character) and then scan for the end of the string.
! Doubled up quotes are reduced to a single quote within the string


```
:12377 12462 2
:12378 12463 2
:12379 12464 2
:12380 12465 2
:12381 12466 2
:12382 12467 2
:12383 12468 2
:12384 12469 2
:12385 12470 2
:12386 12471 2
:12387 12472 2
:12388 12473 2
:12389 12474 2
:12390 12475 2
:12391 12476 4
:12392 12477 4
:12393 12478 4
:12394 12479 4
:12395 12480 2
:12396 12481 2
:12397 12482 2
:12398 12483 2
:12399 12484 2
:12400 12485 2
:12401 12486 2
:12402 12487 2
:12403 12488 2
:12404 12489 2
:12405 12490 2
:12406 12491 2
:12407 12492 2
:12408 12493 2
:12409 12494 2
:12410 12495 2
:12411 12496 2
:12412 12497 2
:12413 12498 2
:12414 12499 2
:12415 12500 2
:12416 12501 2
:12417 12502 2
:12418 12503 2
:12419 12504 2
:12420 12505 2
:12421 12506 2
:12422 12507 2
:12423 12508 2
:12424 12509 2
:12425 12510 2
:12426 12511 2
:12427 12512 1
```

```
! and a carriage-return (end of line) is treated as an error (quotes
! not balanced). Doubled up quotes set DOUBLED_QUOTES flag.
DOUBLED_QUOTES = FALSE;
QUOTE = .CHARPTR[0];
TOKENLEN = 0;
TOKEN_TYPE[0] = TOKEN$K_STRING;

WHILE TRUE DO
  BEGIN
    CHARPTR = .CHARPTR + 1;
    IF .CHARPTR[0] EQL CAR RET THEN SIGNAL(DBG$_MATQUOMIS);
    IF .CHARPTR[0] EQL .QUOTE
    THEN
      BEGIN
        CHARPTR = .CHARPTR + 1;
        IF .CHARPTR[0] NEQ .QUOTE THEN EXITLOOP;
        DOUBLED_QUOTES = TRUE;
      END;

      IF .TOKENLEN GEQ 255 THEN SIGNAL(DBG$_QUOSTRLONG);
      TOKENLEN = .TOKENLEN + 1;
      TOKENBUFFER[.TOKENLEN] = .CHARPTR[0];
    END;

    ! If language is PL/I, then check for a bit-string.
    ! If the quotes are "" and there were no doubled up quotes, then
    ! see if the next character is the letter 'B' or 'b'. If so, then
    ! change TOKEN_TYPE to be a bit-string (TOKEN$K_BIT_STRING). The
    ! 'B' is not part of the length of the string.
    IF .DBG$GB LANGUAGE EQL DBG$K_PLI
      AND .QUOTE EQL DBG$K_QUOTE
      AND NOT .DOUBLED_QUOTES
    THEN
      IF .CHARPTR[0] EQL %C'B' OR .CHARPTR[0] EQL %C'b'
      THEN
        BEGIN
          CHARPTR = .CHARPTR + 1;
          TOKEN_TYPE[0] = TOKEN$K_BIT_STRING;
        END;

        ! We found the end of the string. Complete the Counted ASCII string in
        ! the TOKENBUFFER buffer by filling in the length and return.
        TOKENBUFFER[0] = .TOKENLEN;
        RETURN;
      END;
```

007C 00000 SCAN_QUOTED_STRING:

			56	00000000G	00	9E	00002	.WORD	Save R2,R3,R4,R5,R6		1240
			55	000000000	EF	9E	00009	MOVAB	LIB\$SIGNAL, R6		
					54	D4	00010	MOVAB	CHARPTR, R5		
			53	00	B5	9A	00012	CLRL	DOUBLED_QUOTES		1246
					52	D4	00016	MOVZBL	@CHARPTR, QUOTE		1246
					02	D0	00018	CLRL	TOKENLEN		1246
			08	BC	02	D0	00018	MOVL	#2, @TOKEN_TYPE		1246
					65	D6	0001C	INCL	CHARPTR		1247
					B5	91	0001E	CMPB	@CHARPTR, #13		1247
					09	12	00022	BNEQ	2\$		
				00028E30	8F	DD	00024	PUSHL	#167472		
			66		01	FB	0002A	CALLS	#1, LIB\$SIGNAL		
53	00	B5			00	ED	0002D	CMPZV	#0, #8, @CHARPTR, QUOTE		1247
					0D	12	00033	BNEQ	3\$		
					65	D6	00035	INCL	CHARPTR		1247
53	00	B5			00	ED	00037	CMPZV	#0, #8, @CHARPTR, QUOTE		1247
					1F	12	0003D	BNEQ	5\$		
			54		01	D0	0003F	MOVL	#1, DOUBLED_QUOTES		1247
			8F		52	D1	00042	CMPL	TOKENLEN, #255		1248
					09	19	00049	BLSS	4\$		
				000289DA	8F	DD	0004B	PUSHL	#166362		
			66		01	FB	00051	CALLS	#1, LIB\$SIGNAL		
					52	D6	00054	INCL	TOKENLEN		1248
			04	BC42	00	B5	90	MOVAB	@CHARPTR, @TOKENBUFFER[TOKENLEN]		1248
					BE	11	0005C	BRB	1\$		1247
					00	91	0005E	CMPB	DBG\$GB_LANGUAGE, #5		1249
					1C	12	00065	BNEQ	7\$		
			27		53	D1	00067	CMPL	QUOTE, #39		1249
					17	12	0006A	BNEQ	7\$		
			14		54	E8	0006C	BLBS	DOUBLED_QUOTES, 7\$		1249
			42	8F	00	B5	91	CMPB	@CHARPTR, #66		1249
					07	13	00074	BEQL	6\$		
			62	8F	00	B5	91	CMPB	@CHARPTR, #98		
					06	12	0007B	BNEQ	7\$		
					65	D6	0007D	INCL	CHARPTR		1250
			08	BC	0D	D0	0007F	MOVL	#13, @TOKEN_TYPE		1250
			04	BC	52	90	00083	MOVAB	TOKENLEN, @TOKENBUFFER		1250
					04	00087		RET			1251

; Routine Size: 136 bytes, Routine Base: DBG\$CODE + 4205

:12428 12513 1
:12429 12514 0 END ELUDOM

.EXTRN LIB\$SIGNAL

PSECT SUMMARY

Name	Bytes	Attributes
DBG\$GLOBAL	4	NOVEC, WRT, RD, NOEXE, NOSHR, LCL, REL, CON, PIC, ALIGN(2)
DBG\$OWN	1344	NOVEC, WRT, RD, NOEXE, NOSHR, LCL, REL, CON, PIC, ALIGN(2)
DBG\$PLIT	13389	NOVEC, NOWRT, RD, EXE, SHR, LCL, REL, CON, PIC, ALIGN(0)

: DBG\$CODE 17037 NOVEC,NOWRT, RD , EXE, SHR, LCL, REL, CON, PIC,ALIGN(0)

Library Statistics

File	----- Total	Symbols Loaded	----- Percent	Pages Mapped	Processing Time
_\$255\$DUA28:[SYSLIB]LIB.L32;1	18619	36	0	1000	00:02.0
_\$255\$DUA28:[DEBUG.OBJ]STRUCDEF.L32;1	32	2	6	7	00:00.1
_\$255\$DUA28:[DEBUG.OBJ]DBGLIB.L32;1	1545	490	31	97	00:02.0
_\$255\$DUA28:[DEBUG.OBJ]DSTRECRDS.L32;1	418	125	29	31	00:00.4
_\$255\$DUA28:[DEBUG.OBJ]DBGMSG.L32;1	386	57	14	22	00:00.3
_\$255\$DUA28:[DEBUG.OBJ]DBGGEN.L32;1	150	1	0	12	00:00.3

: Information: 2
: Warnings: 0
: Errors: 0

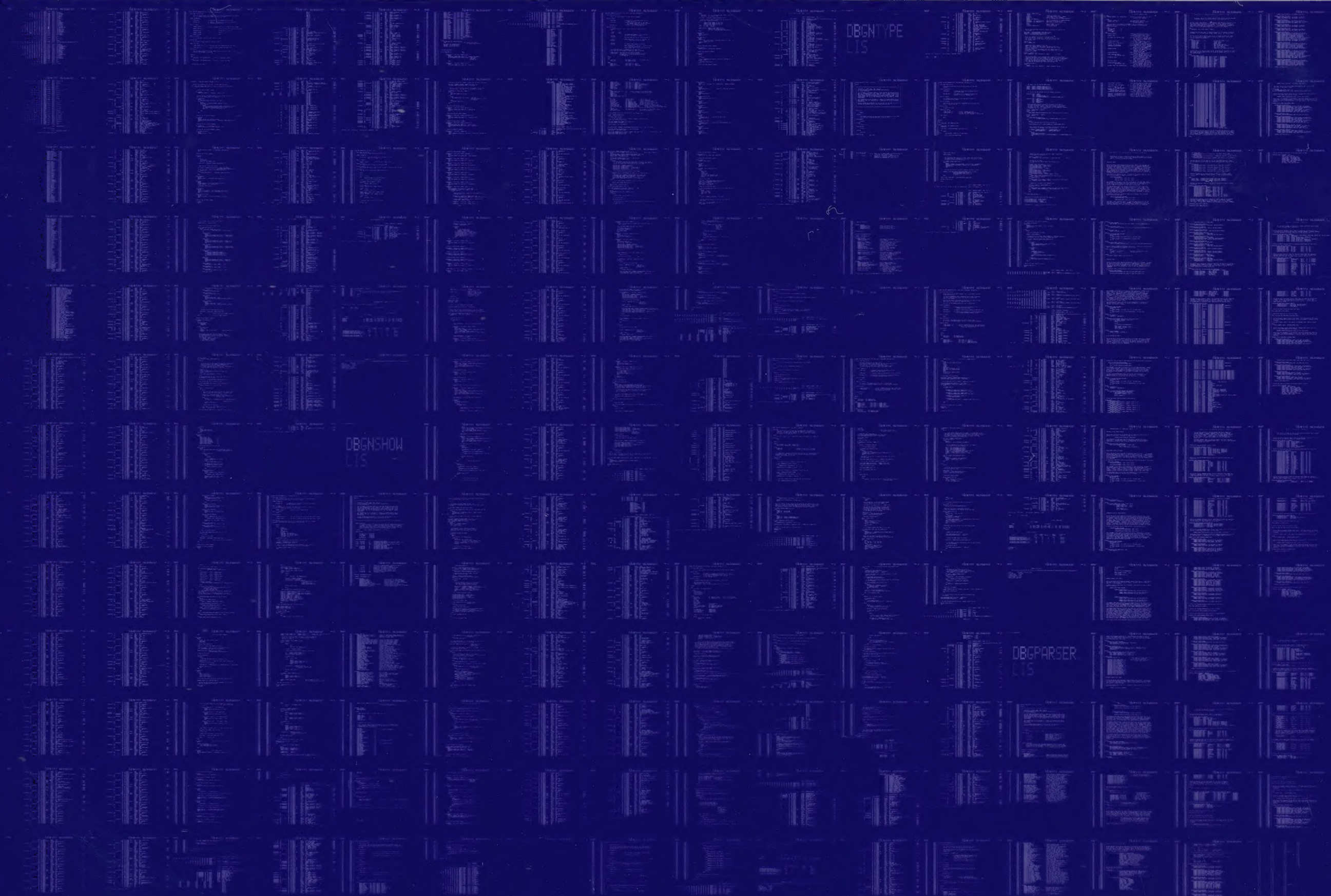
COMMAND QUALIFIERS

: BLISS/CHECK=(FIELD,INITIAL,OPTIMIZE)/LIS=LIS\$:DBGPARSER/OBJ=OBJ\$:DBGPARSER MSRC\$:DBGPARSER/UPDATE=(ENH\$:DBGPARSER)

: Size: 17037 code + 14737 data bytes
: Run Time: 07:23.7
: Elapsed Time: 17:24.9
: Lines/CPU Min: 1692
: Lexemes/CPU-Min: 28123
: Memory Used: 1461 pages
: Compilation Complete

0089 AH-BT13A-SE
VAX/VMS V4.0

DIGITAL EQUIPMENT CORPORATION
CONFIDENTIAL AND PROPRIETARY



AH-BT13A-SE
VAX/VMS V4.0

DIGITAL EQUIPMENT CORPORATION
CONFIDENTIAL AND PROPRIETARY

0091

**DIGITAL
CONFIDENTIAL**